LHT
42390.P10788
Intel

PATE COOPERATION TREATY

From the INTERNATIONAL SEARCHING AUTHORITY

PCT

To:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN
Attn. Mallie, Michael J.
12400 Wilshire Boulevard
7th floor
Los Angeles, CA 90025
UNITED STATES OF AMERICA

NOTIFICATION OF TRANSMITTAL OF THE INTERNATIONAL SEARCH REPORT OR THE DECLARATION

(PCT Rule 44.1)

	Date of mailing *(day/month/year)* 29/10/2002
Applicant's or agent's file reference P10788PCT	**FOR FURTHER ACTION** See paragraphs 1 and 4 below
International application No. PCT/US 02/06292	International filing date *(day/month/year)* 28/02/2002
Applicant INTEL CORPORATION	

1. ☒ The applicant is hereby notified that the International Search Report has been established and is transmitted herewith.

Filing of amendments and statement under Article 19:
The applicant is entitled, if he so wishes, to amend the claims of the International Application (see Rule 46):

When? The time limit for filing such amendments is normally 2 months from the date of transmittal of the International Search Report; however, for more details, see the notes on the accompanying sheet.

Where? Directly to the International Bureau of WIPO
34, chemin des Colombettes
1211 Geneva 20, Switzerland
Fascimile No.: (41-22) 740.14.35

For more detailed instructions, see the notes on the accompanying sheet.

2. □ The applicant is hereby notified that no International Search Report will be established and that the declaration under Article 17(2)(a) to that effect is transmitted herewith.

3. □ **With regard to the protest** against payment of (an) additional fee(s) under Rule 40.2, the applicant is notified that:

□ the protest together with the decision thereon has been transmitted to the International Bureau together with the applicant's request to forward the texts of both the protest and the decision thereon to the designated Offices.

□ no decision has been made yet on the protest; the applicant will be notified as soon as a decision is made.

4. **Further action(s):** The applicant is reminded of the following:

Shortly after **18 months** from the priority date, the international application will be published by the International Bureau. If the applicant wishes to avoid or postpone publication, a notice of withdrawal of the international application, or of the priority claim, must reach the International Bureau as provided in Rules 90*bis*.1 and 90*bis*.3, respectively, before the completion of the technical preparations for international publication.

Within **19 months** from the priority date, a demand for international preliminary examination must be filed if the applicant wishes to postpone the entry into the national phase until 30 months from the priority date (in some Offices even later).

Within **20 months** from the priority date, the applicant must perform the prescribed acts for entry into the national phase before all designated Offices which have not been elected in the demand or in a later election within 19 months from the priority date or could not be elected because they are not bound by Chapter II.

Name and mailing address of the International Searching Authority European Patent Office, P.B. 5818 Patentlaan 2 NL-2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Selwa Harris

Due date 12/29/2002 Client Name Intel Corporation

Docket Initial 42390 . P10788PCT PCT

Dock. Sup. Initial

Atty/Initial EHT SKW

Pat/Ser/Reg US02/06292 32 x

Description

Deadline to submit amended claims to WIPO/Switzerland for annexation to original claims in publication (check to see if search report has been received prior to sending amendment)

Due date 11/29/2002 Client Name Intel Corporation

Docket Initial 42390 . P10788PCT PCT

Dock. Sup. Initial

Atty/Initial EHT SKW

Pat/Ser/Reg US02/06292 32r

Description

Reminder:Deadline to submit amended claims to WIPO/Switzerlan for annexation to original claims in publication (check to see if search report has been received prior to sending amendment) is 12/2

Due date 1/29/2003 Client Name Intel Corporation

Docket Initial 42390 . P10788PCT PCT

Dock. Sup. Initial

Atty/Initial EHT SKD

Pat/Ser/Reg US02/06292 14 x

Description

Deadline to file the information disclosure statement re: PCT search report (in all related US cases)

Natalie Adair

Due date 11/29/2002 Client Name Intel Corporation

Docket Initial 42390 . P10788PCT PCT

Dock. Sup. Initial

Atty/Initial EHT SKD

Pat/Ser/Reg US02/06292 14r

Description

Reminder:Deadline to file the information disclosure statement re: PCT search report (in all related US cases) is 1/29/03

NOTES TO FORM PCT/ISA/220

These Notes are intended to give the basic instructions concerning the filing of amendments under article 19. The Notes are based on the requirements of the Patent Cooperation Treaty, the Regulations and the Administrative Instructions under that Treaty. In case of discrepancy between these Notes and those requirements, the latter are applicable. For more detailed information, see also the PCT Applicant's Guide, a publication of WIPO.

In these Notes, "Article", "Rule", and "Section" refer to the provisions of the PCT, the PCT Regulations and the PCT Administrative Instructions respectively.

INSTRUCTIONS CONCERNING AMENDMENTS UNDER ARTICLE 19

The applicant has, after having received the international search report, one opportunity to amend the claims of the international application. It should however be emphasized that, since all parts of the international application (claims, description and drawings) may be amended during the international preliminary examination procedure, there is usually no need to file amendments of the claims under Article 19 except where, e.g. the applicant wants the latter to be published for the purposes of provisional protection or has another reason for amending the claims before international pbulication. Furthermore, it should be emphasized that provisional protection is available in some States only.

What parts of the international application may be amended?

Under Article 19, only the claims may be amended.

During the international phase, the claims may also be amended (or further amended) under Article 34 before the International Preliminary Examining Authority. The description and drawings may only be amended under Article 34 before the International Examining Authority.

Upon entry into the national phase, all parts of the international application may be amended under Article 28 or, where applicable, Article 41.

When? Within 2 months from the date of transmittal of the international search report or 16 months from the priority date, whichever time limit expires later. It should be noted, however, that the amendments will be considered as having been received on time if they are received by the International Bureau after the expiration of the applicable time limit but before the completion of the technical preparations for international publication (Rule 46.1).

Where not to file the amendments?

The amendments may only be filed with the International Bureau and not with the receiving Office or the International Searching Authority (Rule 46.2).

Where a demand for international preliminary examination has been/is filed, see below.

How? Either by cancelling one or more entire claims, by adding one or more new claims or by amending the text of one or more of the claims as filed.

A replacement sheet must be submitted for each sheet of the claims which, on account of an amendment or amendments, differs from the sheet originally filed.

All the claims appearing on a replacement sheet must be numbered in Arabic numerals. Where a claim is cancelled, no renumbering of the other claims is required. In all cases where claims are renumbered, they must be renumbered consecutively (Administrative Instructions, Section 205(b)).

The amendments must be made in the language in which the international application is to be published.

What documents must/may accompany the amendments?

Letter (Section 205(b)):

The amendments must be submitted with a letter.

The letter will not be published with the international application and the amended claims. It should not be confused with the "Statement under Article 19(1)" (see below, under "Statement under Article 19(1)").

The letter must be in English or French, at the choice of the applicant. However, if the language of the international application is English, the letter must be in English; if the language of the international application is French, the letter must be in French.

NOTES TO FORM PCT/ISA/220 (continued)

The letter must indicate the differences between the claims as filed and the claims as amended. It must, in particular, indicate, in connection with each claim appearing in the international application (it being understood that identical indications concerning several claims may be grouped),whether

(i) the claim is unchanged;

(ii) the claim is cancelled;

(iii) the claim is new;

(iv) the claim replaces one or more claims as filed;

(v) the claim is the result of the division of a claim as filed.

The following examples illustrate the manner in which amendments must be explained in the accompanying letter:

1. [Where originally there were 48 claims and after amendment of some claims there are 51]: "Claims 1 to 29, 31, 32, 34, 35, 37 to 48 replaced by amended claims bearing the same numbers; claims 30, 33 and 36 unchanged; new claims 49 to 51 added."
2. [Where originally there were 15 claims and after amendment of all claims there are 11]: "Claims 1 to 15 replaced by amended claims 1 to 11."
3. [Where originally there were 14 claims and the amendments consist in cancelling some claims and in adding new claims]: "Claims 1 to 6 and 14 unchanged; claims 7 to 13 cancelled; new claims 15, 16 and 17 added." or "Claims 7 to 13 cancelled; new claims 15, 16 and 17 added; all other claims unchanged."
4. [Where various kinds of amendments are made]: "Claims 1-10 unchanged; claims 11 to 13, 18 and 19 cancelled; claims 14, 15 and 16 replaced by amended claim 14; claim 17 subdivided into amended claims 15, 16 and 17; new claims 20 and 21 added."

"Statement under article 19(1)" (Rule 46.4)

The amendments may be accompanied by a statement explaining the amendments and indicating any impact that such amendments might have on the description and the drawings (which cannot be amended under Article 19(1)).

The statement will be published with the international application and the amended claims.

It must be in the language in which the international appplication is to be published.

It must be brief, not exceeding 500 words if in English or if translated into English.

It should not be confused with and does not replace the letter indicating the differences between the claims as filed and as amended. It must be filed on a separate sheet and must be identified as such by a heading, preferably by using the words "Statement under Article 19(1)."

It may not contain any disparaging comments on the international search report or the relevance of citations contained in that report. Reference to citations, relevant to a given claim, contained in the international search report may be made only in connection with an amendment of that claim.

Consequence if a demand for international preliminary examination has already been filed

If, at the time of filing any amendments under Article 19, a demand for international preliminary examination has already been submitted, the applicant must preferably, at the same time of filing the amendments with the International Bureau, also file a copy of such amendments with the International Preliminary Examining Authority (see Rule 62.2(a), first sentence).

Consequence with regard to translation of the international application for entry into the national phase

The applicant's attention is drawn to the fact that, where upon entry into the national phase, a translation of the claims as amended under Article 19 may have to be furnished to the designated/elected Offices, instead of, or in addition to, the translation of the claims as filed.

For further details on the requirements of each designated/elected Office, see Volume II of the PCT Applicant's Guide.

PATENT COOPERATION TREATY

PCT

INTERNATIONAL SEARCH REPORT

(PCT Article 18 and Rules 43 and 44)

Applicant's or agent's file reference P10788PCT	**FOR FURTHER ACTION** see Notification of Transmittal of International Search Report (Form PCT/ISA/220) as well as, where applicable, item 5 below.	
International application No. PCT/US 02/ 06292	International filing date *(day/month/year)* 28/02/2002	(Earliest) Priority Date *(day/month/year)* 26/03/2001
Applicant INTEL CORPORATION		

This International Search Report has been prepared by this International Searching Authority and is transmitted to the applicant according to Article 18. A copy is being transmitted to the International Bureau.

This International Search Report consists of a total of 4 sheets.

☒ It is also accompanied by a copy of each prior art document cited in this report.

1. **Basis of the report**

a. With regard to the **language**, the international search was carried out on the basis of the international application in the language in which it was filed, unless otherwise indicated under this item.

□ the international search was carried out on the basis of a translation of the international application furnished to this Authority (Rule 23.1(b)).

b. With regard to any **nucleotide and/or amino acid sequence** disclosed in the international application, the international search was carried out on the basis of the sequence listing :

□ contained in the international application in written form.

□ filed together with the international application in computer readable form.

□ furnished subsequently to this Authority in written form.

□ furnished subsequently to this Authority in computer readble form.

□ the statement that the subsequently furnished written sequence listing does not go beyond the disclosure in the international application as filed has been furnished.

□ the statement that the information recorded in computer readable form is identical to the written sequence listing has been furnished

2. □ **Certain claims were found unsearchable** (See Box I).

3. □ **Unity of invention is lacking** (see Box II).

4. With regard to the **title**,

☒ the text is approved as submitted by the applicant.

□ the text has been established by this Authority to read as follows:

NO DOCKETING REQUIRED
AD

5. With regard to the **abstract**,

☒ the text is approved as submitted by the applicant.

□ the text has been established, according to Rule 38.2(b), by this Authority as it appears in Box III. The applicant may, within one month from the date of mailing of this international search report, submit comments to this Authority.

6. The figure of the **drawings** to be published with the abstract is Figure No. 2

□ as suggested by the applicant.

☒ because the applicant failed to suggest a figure.

□ because this figure better characterizes the invention.

□ None of the figures.

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 02/06292

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F11/36

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, INSPEC, WPI Data, PAJ

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	ZHANG X ET AL: "SYSTEM SUPPORT FOR AUTOMATIC PROFILING AND OPTIMIZATION" OPERATING SYSTEMS REVIEW (SIGOPS), ACM HEADQUARTER. NEW YORK, US, vol. 31, no. 5, 1 December 1997 (1997-12-01), pages 15-26, XP000771018 cited in the application	1,19
A	abstract paragraph '0003! paragraph '0005!, sentence 2 ---	2-18, 20-27

-/--

☒ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search 10 October 2002	Date of mailing of the international search report 29/10/2002
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Renault, S

1

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 02/06292

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
X	MERTEN M C ET AL: "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization" COMPUTER ARCHITECTURE, 1999. PROCEEDINGS OF THE 26TH INTERNATIONAL SYMPOSIUM ON ATLANTA, GA, USA 2-4 MAY 1999, LOS ALAMITOS, CA, USA,IEEE COMPUT. SOC, US, 2 May 1999 (1999-05-02), pages 136-148, XP010334956 ISBN: 0-7695-0170-2	1,19
A	abstract paragraph ‘0002! ---	2-18, 20-27
A	CONTE T M ET AL: "USING BRANCH HANDLING HARDWARE TO SUPPORT PROFILE-DRIVEN OPTIMIZATION" PROCEEDINGS OF THE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, XX, XX, 30 November 1994 (1994-11-30), pages 12-21, XP000783845 paragraph ‘0003! ---	1-27
A	BURGER R G ET AL: "AN INFRASTRUTURE FOR PROFILE-DRIVEN DYNAMIC RECOMPILATION" PROCEEDINGS OF THE 1998 INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES. ICCL '98. CHICAGO, IL, MAY 14 - 16, 1998, INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES, LOS ALAMITOS, CA: IEEE COMPUTER SOC, US, 14 May 1998 (1998-05-14), pages 240-249, XP000883603 ISBN: 0-7803-5005-7 paragraphs ‘0001!-‘0003! ---	1-27
A	EP 0 999 499 A (HEWLETT PACKARD CO) 10 May 2000 (2000-05-10) abstract paragraphs ‘0074!,‘0075! -----	1-27

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 02/06292

Patent document cited in search report		Publication date	Patent family member(s)			Publication date
EP 0999499	A	10-05-2000	US	6233678	B1	15-05-2001
			EP	0999499	A2	10-05-2000
			JP	2000148482	A	30-05-2000

System Support for Automatic Profiling and Optimization

V 8283

p. 15-26 = 12

Xiaolan Zhang, Zheng Wang, Nicholas Gloy,
J. Bradley Chen, and Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University

XP-000771018

p d. 11-1997

Abstract

The Morph system provides a framework for automatic collection and management of profile information and application of profile-driven optimizations. In this paper, we focus on the operating system support that is required to collect and manage profile information on an end-user's workstation in an automatic, continuous, and transparent manner. Our implementation for a Digital Alpha machine running Digital UNIX 4.0 achieves run-time overheads of less than 0.3% during profile collection. Through the application of three code layout optimizations, we further show that Morph can use statistical profiles to improve application performance. With appropriate system support, automatic profiling and optimization is both possible and effective.

1. Introduction

Morph is a combination of operating system and compiler technology that provides a practical framework for the advanced compiler optimizations needed to support continued improvements in application performance. Morph is practical because it provides system support for the automatic application of profile-driven compiler optimizations on an end-user's system. The major obstacle to the use of these optimizations is obtaining timely high-quality profile information. Ideally, such profiles should be representative of how a program is used on a specific machine by a specific user. Given the current infrastructure for compilation, it is not practical to obtain and use such profiles, as typical end-users do not know how to generate profile information and would not be able to optimize the application if they did. The Morph system addresses this problem, using operating system support to collect, process, and apply profile information to re-optimize applications automatically.

This paper describes our implementation of Morph for the Digital Alpha processor and Digital UNIX. Our implementation is composed of several components. The *Morph Monitor* is an operating system kernel component that implements continuous, low overhead profiling and program monitoring. The *Morph Editor* is a compiler component that implements re-optimization, transforming compiler intermediate form into an executable. The *Morph Manager* is a system component that manages profile information, including the automatic invocation of re-optimization. In this paper we focus on the systems components, the Monitor and the Manager. Although some discussion of the Morph Editor is needed to provide context for the rest of the work, detailed description of specific compiler optimizations is outside the scope of this paper.

We designed our system to meet a number of requirements that we believe are necessary for mainstream systems and users:

- *Optimization should happen on the machine where the software is used.* For complex interactive applications, there can be substantial variation in how the application is used by different individuals. Also, binaries for popular applications run on a broad range of binary-compatible systems with widely varying internal architecture and performance characteristics. Given these sources of variation between systems, profile-driven optimization will be most effective if it can incorporate information specific to the end-user and the user's machine.
- *Optimization should not require source code.* Although recompilation from source code may be a reasonable alternative for public-domain and low-volume software, it is not practical for the commodity software market. To enable optimization without source code, we assume that an intermediate-form representation of the program can be derived from the program distribution media, with all optimizations based on this intermediate form. In practice, there are a number of alternatives for achieving this, including executable editing techniques [LS95, RVL97] and compact program representations [FK96].
- *Optimization must be transparent to the user.* We assume that the people who use computers are non-specialists with no background in computer science or compilation. Optimization should be transparent in that the user does not need to understand or actively participate in the optimization process. Transparency also implies that the system be robust and provide consistent performance benefits with no negative performance impact.

The Morph system is designed to address all these requirements. It automatically collects profiles and uses them to optimize programs on the machine where the software is used. The optimizations transform an intermediate form representation of the program to a new executable without requiring source code. As an intermediate form representation is not shipped with current applications, it must be generated from source in our system. Morph makes it possible to re-optimize an application using a small number of simple,

SOSP-16 10/97 Saint-Malo, France
© 1997 ACM 0-89791-916-5/97/0010...$3.50

mechanical steps. Our results show that Morph is an effective platform for profile-based optimization. We also show that the overhead of Morph profile collection and management is negligible. This paper describes a design and implementation of Morph, and gives performance results that illustrate the measured benefit of profile-driven optimization on our testbed system.

1.1 Related Work

There are a number of research projects in low-overhead profiling and late optimization. In this section we review some relevant projects and explain how they differ from Morph.

Morph enables executable programs to evolve with changes in program usage patterns and computer hardware. This functionality is not provided by any current tool, although Morph builds on technology and techniques originally developed for program instrumentation and measurement. Several existing tools (Pixie [Smi91], ATOM [SE94], EEL [LS95]) rewrite executables for the purpose of program measurement. These instrumentation tools differ from Morph in two important ways. First, though the overhead of instrumention-based profiling has improved, yielding slowdowns as low as 10%, this level of overhead is still too large for continuous monitoring, frequently exceeding the benefit of optimization. In Morph, we avoid this overhead through statistical profile sampling in the operating system. In Section 4.3, we show that the on-line overhead for sample collection in Morph is significantly less than 1%. A further difference between Morph and earlier profiling tools is that Morph is designed specifically for automatic profiling and optimization.

Morph applies profile-driven, machine-specific optimizations to intermediate representations of programs and libraries. Like Morph, three prior systems from Digital Equipment Corporation (Mahler [WP87], Epoxie [Wal92], OM [SW92]) also performed optimizations on programs after compilation. These tools were designed for UNIX systems and required either an intermediate form representation of the program or an executable that includes relocation information. More recent systems for Windows NT include Spike [CL96, Goo97] and Etch [RVL97]. Both implement profile-based optimizations, Etch for the Intel x86 architecture and Spike for Digital Alpha-based systems.

The Digital Continuous Profiling Infrastructure (DCPI), developed at the Digital Systems Research Center, supports continuous monitoring of the entire system including the kernel [ABD97]. There are substantial similarities between DCPI profile collection and the continuous monitor used in Morph. Both DCPI and the Morph Monitor use statistical sampling to collect profile information of all system activity with very low overhead. The primary difference between the two monitors is how they have been applied. In Morph we have focused on optimization, whereas the emphasis to date for DCPI has been on performance analysis. It would be relatively straightforward to use DCPI as a source of profiles for Morph.

Conte et al. [CMH96] describe an approach in which new hardware support in the form of a profile buffer is used to collect statistics describing the behavior of conditional branches in the program. They use the resulting information to drive a superblock scheduling optimization. Although the Morph design does not preclude the use of new hardware support for profile collection, it does not require it either.

Digital's FX!32 is a system that provides transparent execution of x86 Win32 applications on Windows NT Alpha systems [CH97, Rub96]. As a part of their emulation system, execution profiles of x86 code are collected and fed to a background process that translates the previously emulated portions of x86 binaries into native Alpha code. Though the goal of Digital FX!32 is different from Morph, the two systems share many similarities: both systems continuously collect profile information of executables in a transparent manner and use the profiles to guide a code re-writing operation. The main differences between the two systems are that FX!32 works directly on executables, it collects profile samples during program emulation (rather than direct execution) of the application, and that it emphasizes code translation, rather than code optimization as in Morph.

In the next section, we discuss the design goals for Morph. Section 3 then describes our implementation. In Section 4, we report performance results, both in terms of optimization and overhead. Section 5 discusses future research directions and other issues in making the system practical for widespread use. Conclusions follow in Section 6.

2. Motivation for System Design

Morph provides a framework for profile-based and host-specific optimization. An important feature of Morph is that optimization occurs on the system where the applications are run. The final stage of optimization occurs after the end-user has already installed and used the application. This makes it possible for optimization to incorporate the idiosyncratic usage patterns of a specific user and track changes in the way a program is used. It also means that all details of the host hardware can be used during optimization. Several observations about profile data and machine description information serve to clarify the potential advantages of the late application of host-specific optimizations.

Profile information describes how an application is used by an end-user. The trend in modern applications is to implement a large number of features and to hide the complexity of feature selection behind a rich graphical user interface. For such applications, the subset of features used by different individuals can vary substantially. This will tend to increase the need for profile-driven optimization and the value of collecting profile information as close as possible to the end-user. Also, the way an individual uses a complex interactive application can evolve over time. For example, a user may discover a new feature in a user-interface and make regular use of it from that time on. This suggests a situation in which profile-based optimizations must be continuously re-applied in order to achieve their full benefit. This variation may not be significant for batch-style UNIX workloads such as those used in this paper. Nonetheless, the techniques we present facilitate exploration of these issues for more complex interactive workloads.

The second type of information used by host-specific optimizations describes a specific implementation of a machine architecture. Many modern compiler optimizations depend on information that is specific to an implementation of an instruction set architecture and is not expressed in the instruction-set description, such as:

- *Pipeline organization*. This information directs instruction scheduling.
- *Instruction timing*. Often, the relative cost of an instruction sequence changes from one machine implementation to the next. This information influences the instruction selection and scheduling processes.
- *Cache parameters*. This information can be used by locality optimizations such as data blocking [LRW91] and code layout [PH90].

These hardware parameters commonly vary between binary-compatible systems, even between those designed at the same time in similar VLSI technologies. In the absence of a suitable infrastructure for host-specific optimization, hardware designers have created machines that attempt to execute programs efficiently without help from the compiler. An example is the current trend towards out-of-order instruction execution, in which the hardware does instruction scheduling on the fly rather than relying on software scheduling. While this has the advantage of providing performance improvements for legacy software, it has the disadvantages of increased hardware complexity and sub-optimal overall performance. In general, the lack of a practical infrastructure for host-specific optimization leads to increased hardware complexity, even when simpler hardware alternatives that require minimal compiler support exist. In spite of the lack of a workable compiler infrastructure, the need for host-specific optimizations is increasing in modern machines. Experts in the computer architecture and compiler communities anticipate that the importance of these optimizations will increase for the next generation of machines [Chr96, Gwe96].

3. Implementation

Our prototype implementation of Morph supports automatic profile generation and re-optimization for Digital Alpha-based workstations running Digital UNIX. We employed several custom system tools and made small modifications to Digital UNIX to support automatic profile generation, collection, and analysis. The compiler components of Morph are based on the SUIF research compiler infrastructure available from Stanford University [SUIF94]. We have extended this infrastructure to support profile-driven, machine-specific optimizations [Smi96]. We are planning support for Windows NT and the x86 ISA, as a step towards optimization of more complex interactive applications. Practical considerations led us to choose the UNIX/Alpha platform for our initial implementation, including access to source code for Digital UNIX. In Section 3.1, we introduce the components of the Morph system and describe how they are incorporated into a complete system for automatic program optimization. The subsections that follow focus on the main systems components of Morph, namely the Morph Monitor for profile sample collection (Section 3.2), the Morph Manager that provides off-line profile processing (Section 3.3), and the Morph Editor that implements optimizations (Section 3.4).

3.1. Overview

Our initial design targets a single-user workstation environment. Specifically, we assume that each user has his or her own machine, and that all instructions executed on the machine are run on behalf of that user. We assume that each user has a private copy of the applications they use. Finally, we assume that there are substantial periods of idle time. These assumptions are appropriate for most workstations and personal computers. For machines that do not satisfy these assumptions, additional engineering is required. For example, in an environment where users share a single copy of an executable file from a file server, the optimized version of the module could either be created on a per-user basis or be shared by all users. The former requires more disk space and the latter requires distributed profile collection. Both require system support comparable to that described in this paper.

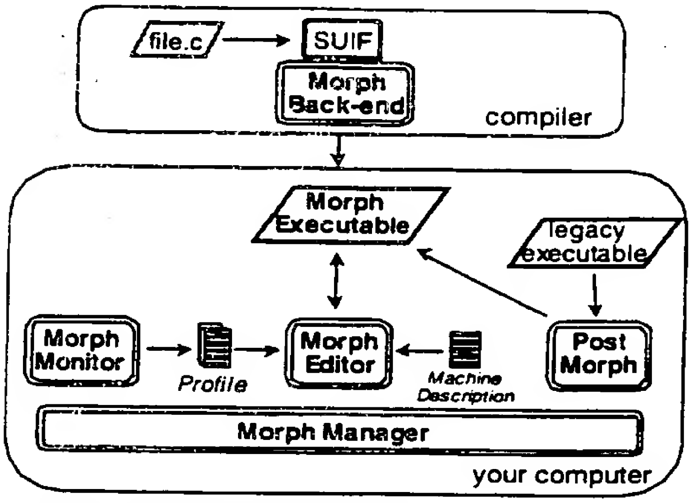


Figure 1. The Structure of Morph.

The major components of Morph are represented schematically in Figure 1. Optimizations in Morph are applied without the use of source code. To achieve this goal we posit an application distribution format that includes a compact representation of the compiler intermediate form. Compiler optimizations are implemented as SUIF passes which apply transformations to this intermediate form. An important aspect of our design is that executable modules are machine and OS specific. Though we are aware of prior work on machine and operating system-neutral program distribution formats [OSF93, Col95], we have chosen not to attempt similar functionality in Morph, focusing instead on the problem of using profile-driven optimizations to improve performance.

The key to providing a viable framework for host-specific optimization is to define a partitioning of the compilation process that supports efficient and effective retargeting. The link between the two phases of compilation is the Morph executable, an executable that includes the supplementary information required for the second, host-specific stage of compilation. Conceptually, a Morph executable consists of two parts. The first part is equivalent to executables in today's systems. This part could be directly executed on the target hardware. The second part, specific to the Morph system, contains the extra information necessary to re-optimize the executable. It includes information that allows recovery of the program intermediate form, as well as the additional Morph annotations required to support robust and general executable rewriting. In our prototype, we maintain these two components in separate files, one for the executable program, and another containing the complete intermediate form representation of the program.

Morph has five major software components: the Morph Back-end, the Morph Manager, the Morph Editor, PostMorph, and the Morph Monitor. The Morph Back-end produces executable programs and shared libraries that include the annotations Morph needs to support efficient late optimization. The Morph Manager is a user-level daemon that provides off-line profile management as well as profile analysis to decide when to invoke the Morph Editor and PostMorph. The Morph Editor implements the optimizations provided by Morph. The current version of our Editor supports three profile-based optimizations. Two of the optimizations, procedure layout and "fluff removal", are designed to improve locality in the instruction reference stream [PH90]. The third is a basic block ordering optimization that improves both branch prediction and instruction reference locality [PH90, YJK97]. The Morph Monitor collects profile information for use by the Morph Editor. The Monitor collects profile samples with very low overhead, leaving subsequent processing to the Morph Manager. PostMorph provides a means for extending the life of legacy executables by inferring Morph annotations through static and dynamic analysis so they can be retargeted by the Morph Editor.

3.2 On-line Profile Collection: The Morph M nitor

Profile collection is implemented as a two-phase process: on-line sample collection in the Morph Monitor, followed by off-line profile analysis and management in the Morph Manager. This separation allows us to minimize the on-line overhead of profile collection by deferring as much work as possible until off-line processing. In this section we describe the design of the Morph Monitor and some of the interesting problems we encountered in its implementation.

One key goal of the Morph Monitor is low profiling overhead. Our monitor achieves this goal by using statistical sampling of program activity, rather than collecting a complete profile. In most current systems, profile information is typically obtained by executing an instrumented version of the application that generates profile information as a side-effect of program execution [Smi91, SE94, LS95]. A straightforward approach for automatic profile collection would be periodic use of a higher-overhead profiling system, such as an instrumentation-based profiler. Although this approach has the advantage that it is well understood, the performance overhead incurred during profiled runs of application would be substantial and in many cases would be perceptible by users. Also, if profile collection were to occur only part of the time, it would be difficult to insure that the resulting profiles were representative of typical program use. By collecting profile information continuously for all activity on the system, we assure that the profiles accurately reflect how the software is used.

The Monitor is implemented as a pseudo-device and a small number of local modifications to the Digital UNIX 4.0 kernel. Profile samples are collected using a modified version of the clock interrupt routine, *hardclock()*, which records a program counter (PC) sample to a sample buffer on each clock interrupt. Idle-loop activity is automatically recognized and eliminated during profile collection. Several additional kernel modifications provide supplementary information needed to interpret the profile samples. We modified the *exec()* system call to provide information about the initialization of address spaces and to selectively enable and disable sampling. We also modified the *mmap()* system call, which is used by the user-level runtime to map executable files and shared libraries into a user address space. We modified the *exit()* system call to log process termination events. This makes it possible to identify when an address space has terminated and the corresponding mapping information can be discarded. We also modified the kernel to log context switch information so that we can interpret each PC sample in the context of the address space in which it was generated.

The kernel sample buffer is statically allocated and is 256 KB in our current system. Periodically, the contents are transferred to the Morph Manager using a UNIX pseudo-device. With 8 bytes per sample and a clock interrupt rate of 1024 Hz, this allows for approximately 30 seconds of profile samples between daemon invocations. The sample buffer is organized as a ring buffer with sample collection continuing during the dump of the sample buffer to disk.

Our current monitor presents a number of opportunities for tuning which we have not yet exploited. These include reducing the number of bytes required to store a profile sample and adjusting the tradeoff between sample buffer size and the frequency of writing samples to disk. The overhead of our current profiler satisfies our main subjective performance goal: it is unnoticeable. Measurements of our current Monitor implementation show the overhead of profiling is typically only about 0.1-0.3% (Section 4.3). Given this level of performance, we have chosen to defer further work on reducing profiling overhead and focus on the other functionality required for automatic optimization.

A potential pitfall of sampling based on clock interrupts is that activity synchronized with clock interrupts might be missed. This problem could be avoided by introducing randomness into the sampling interval, using an interrupt based on a countdown timer instead of (or in addition to) sampling during clock interrupts. This scheme is used by the continuous profiler in DCPI [ABD97]. Many popular processors provide appropriate hardware support, including the Digital Alpha and the Intel Pentium Pro. Using a version of the Morph Monitor that supports random sampling intervals, we determined that sampling on clock interrupts is appropriate for the benchmarks used in this paper.

3.3 Off-line Profile Processing: The M rph Manager

The Morph Manager is implemented as a user-level daemon that periodically reads and processes the raw profile samples generated by the Monitor. Whereas the Monitor is responsible for on-line profile sample collection, the Manager performs off-line profile management and processing. The goals of this processing are twofold: to decide when to invoke an optimization and to transform the raw PC samples into a compact form that can be used directly by optimizations. In our work to date we have focused on transforming profile data for use by optimizations.

From the point of view of optimization, applications are composed of multiple modules, typically with one executable file and multiple dynamically linked libraries (DLLs). Each use of a module results in some number (possibly zero) of Monitor samples. We use the term *sample set* to refer to the PC samples resulting from activity in a single module for a single process. A single sample set may not include enough information to achieve the full potential benefit of a given optimization. This makes it desirable to collect and combine multiple sample sets into a single composite profile for the module.

The Manager identifies executable modules in the profile sample log by their path-name in the file system. If the corresponding file is updated for some other reason than re-optimization (for example, a program update), the system needs to recognize this event, so that profile samples can be interpreted in the correct context. A straightforward scheme for implementing this is to discard profile samples collected during a period in which the corresponding module was updated. A more sophisticated scheme could use a cryptographically secure checksum of the module, that would be recorded in the profile log and checked to verify the correspondence between the module that generated the profile samples and the module currently on disk. We expect that the simple scheme will be adequate in most situations.

program code			statistical profile	
	complete profile	sample set	without scaling	with scaling
`Loop:`				
`. . .`				
`block_A:`			3	1
`mull t1, t1, t2`	1000	2		
`subq t2, 0xa, t2`	1000	0		
`blt t2, block_C`	1000	1		
`block_B:`			6	1
`addq a1, 0x8, a1`	1000	0		
`addq a2, 0x8, a2`	1000	1		
`ldq t3, 0(a1)`	1000	2		
`ldq t4, 0(a2)`	1000	2		
`subq t3, t4, t3`	1000	0		
`addq t0, t3, t0`	1000	1		
`block_C:`				
`. . .`				
`jmp Loop`				

Figure 2: Sample scaling for two basic blocks.

In the conversion from profile sample sets to a basic block profile, each PC sample is a witness for the basic block containing it. Morph provides the information needed to identify basic block boundaries in terms of instruction addresses and to map these address ranges to basic blocks in the intermediate form representation of the executable module. This mapping information is required for the use of profiles based on PC values, such as those obtained through statistical sampling.

The PC samples must be scaled when incrementing the counters for basic blocks. To understand the need for scaling, consider two basic blocks, one containing three instructions, one containing six instructions, and both of which are executed the same number of times during a process. Figure 2 illustrates this example. When processing profile samples, the Manager assumes two instructions that are executed the same number of times have the same probability of being sampled. If we increment the per-basic block counter by one for each sample, the resulting profile would indicate (incorrectly) that block B was executed twice as often as block A. Scaling the increment by the inverse of the basic block length gives better correlation between the weights in the profile and the number of times the basic block was executed. Differences in the cycles required to execute individual instructions also affect the profiles. We will discuss these effects shortly.

After transforming PC samples into a basic block profile, we combine the individual profiles from each run to generate a single basic block profile. The Manager sums the counts from each program execution for each basic block to create an aggregate profile. This method of combining sample sets models the actual usage of the application, as each run of the application is represented in the cumulative profile in proportion to the amount of execution time it required. During this conversion, profile samples for modules that have been modified or deleted are eliminated.

The combining of profile data is a general problem for profile-based optimization and is not unique to our methodology. In conventional (non-automatic) profile-based optimization, complete profiles are collected for a number of scripted training input sets. This is in contrast to continuous profiling of all activity as with Morph. Fisher and Freudenberger [FF92] evaluate three techniques for combining multiple branch profiles: combining profiles directly, normalizing to give each test case the same weight in the composite profile, and polling[1]. The Morph Manager uses direct combining, such that profiles contribute to the composite profile in proportion to their contribution to overall activity. We believe that direct combining is appropriate for a continuous monitor, even though normalization can give better results for scripted inputs [FF92].

When a module is optimized, the profile information for the module must also be transformed, to make it meaningful in the context of the new executable module. As our profiles are stored in terms of basic blocks in the program's intermediate form representation, this transformation requires a function that maps basic blocks in the old intermediate representation to basic blocks in the new intermediate representation. In Morph this mapping function must be generated as a by-product of optimization.

Our profiles tend to give more weight to basic blocks with higher latency and higher average cycles per instruction (CPI), as they have a higher relative probability of being sampled; we call these *time-based profiles*. In contrast,

[1] Fisher and Freudenberger found that polling gave inferior results. This technique will not be discussed further.

profiles generated by instrumented program execution give equal weight to all instructions, regardless of their relative latency; we refer to these as *frequency-based profiles*. Because of this difference, time-based profiles and frequency-based profiles of the same activity are not identical. We could attempt to adjust the time base in our statistical profiles to make them more similar to a frequency-based profile. However, our experience to date suggests that neither time nor frequency based profiles are clearly superior. [GWZ97]

3.4 The Morph Editor

The Morph Editor applies optimizations using profiles from the Morph Monitor. The Editor is currently implemented as a composition of SUIF compiler passes which convert a program module in a low-SUIF intermediate form to optimized assembly language. For this paper, the Editor performed three profile-driven code layout optimizations. The input to the Editor has already been highly optimized using a set of typical scalar optimizations including dead-code removal, as well as machine-specific optimizations such as register allocation. The rest of this section provides a brief description of the optimizations implemented by the Editor. All the optimizations require machine-specific information and benefit from profile information, including basic block execution counts and control flow graph edge frequency counts. The Morph Monitor directly generates basic block counts, and the Editor analyzes these counts to synthesize the edge frequencies. The counts and frequencies do not have to be exact since the information is used only to direct optimization toward the most popular parts of the program. Since these optimizations make some paths through the application run faster at the expense of others, they perform better when profile information can be used in place of simple global heuristics (e.g. assuming forward branches taken and backward branches not-taken).

The three code layout optimizations implemented in the Editor are branch alignment, fluff removal, and procedure layout. All are based on previous work by Pettis and Hansen [PH90]. The first optimization is branch alignment. This intra-procedural optimization reorders the program basic blocks to reduce control penalties, branch mispredictions and misfetches, and to improve instruction cache locality. We implement a greedy version of this technique, described by Young et al. [YJK97]. The algorithm considers the edge frequencies in order of decreasing weight and attempts to place the blocks at the ends of an edge adjacent to each other so that the control penalties are minimized. Fluff removal follows branch alignment. This optimization uses the basic block counts to identify blocks of code (such as error handling code) that were not executed during the profiled run. Moving these blocks to the end of the text segment improves the density (spatial locality) of the remaining code. The relocation of fluff code requires additional jumps to insure that the code is still reachable[2].

Once we have compacted the executed portion of each procedure, we apply a procedure layout optimization to find a placement of the program procedures in memory that minimizes the penalty cycles due to instruction cache conflict and capacity misses. We place all of the non-fluff procedure components before placing all of the fluff components. In particular, our implementation constructs a weighted call graph from the block counts and uses it to identify procedure conflicts that incur the largest cache penalties. Our greedy algorithm attempts to minimize penalties by using a closest-is-best heuristic: if two procedures call each other frequently, the algorithm attempts to place them near each other in the code segment.

4. Experimental Results

This section describes our experiments to evaluate automatic profile-based optimization under Morph. After describing the experimental system and workloads, we present two sets of results. One set of experiments (Section 4.2) documents the effectiveness of profile-based optimization under Morph. In the second set of experiments (Section 4.3), we quantify and analyze the overheads in Morph.

Overall our results show that the optimizations applied by Morph achieve substantial performance improvements for many of our test workloads. Though the focus of this paper is not to explore the effectiveness of such optimizations, these results are important as justification for providing support for automatic optimization as part of an operating system. However, readers should keep in mind that the optimizations used in Morph are well understood by the compiler community, and the main criteria upon which Morph should be evaluated is its effectiveness as a framework for automatic optimization.

4.1 Experimental Details

Our Morph prototype system used version 1.1.2 of the Stanford SUIF compiler with Harvard Machine SUIF extensions (version 1.1.2). We ran our experiments on an AlphaStation 400 4/233. The system includes a 512 KB second-level cache and 128 MB of main memory. Profile information was collected by sampling the program counter on every clock interrupt, producing 1024 samples per second.

Our experimental system is based on Digital UNIX version 4.0. The Morph Monitor is implemented as a set of modifications to the Digital UNIX kernel, with other Morph components running within the infrastructure provided by Digital UNIX.

[2] There is clearly an interaction between instruction scheduling, branch alignment, and fluff removal. Managing this interaction is an area of on-going research in the compiler community.

Benchmark		*Description*	*Text (KB)*
public license	gzip	file compression utility	96
	mpeg_play	video file player	248
Digital UNIX	sort	data file sorting or merging utility	56
Spec92	espresso	boolean function minimization	248
Spec95	go	oriental board game	416
	ijpeg	JPEG image compression and decompression	248
	lisp	LISP language interpreter	112
	perl	Perl language interpreter	488
	vortex	object-oriented database transaction system	800

Table 1. Experimental Workloads.

Benchmark	*Testing Input*	*Training Input*
espresso	*tial.in* from Spec92	seven other input files from Spec92, repeated five times.
go	test input from Spec95	ref and train inputs from Spec95
gzip	one 6.43MB file	ten other files with size between 0.47MB-11.7MB
ijpeg	train input from Spec95	ref and test inputs from Spec95
lisp	test input from Spec95 (8-queen problem)	ref and train inputs from Spec95
mpeg_play	one video file downloaded from the Internet	ten other video files downloaded from the Internet
perl	test input (*jumble.pl*) from Spec95	ref input (*primes.pl* and *scrabbl.pl*) from Spec95
sort	sort a 0.75MB data file	ten other sorting tasks
vortex	train input from Spec95	test input from Spec95

Table 2. Testing and Training inputs.

	% of total activity			*Time (in seconds)*	
Program	*app*	*lib*	*kernel*	*test*	*train*
espresso	90.7	8.4	0.9	11.3	23.8
go	99.7	0.2	0.1	354.7	1917.7
gzip	97.5	0.3	2.2	25.5	184.7
ijpeg	98.5	1.3	0.1	20.6	1284.0
lisp	98.9	0.9	0.3	17.8	1009.5
mpeg_play	93.5	5.6	0.8	18.7	99.2
perl	83.5	12.1	4.4	43.7	799.5
sort	53.6	41.8	4.6	13.4	79.8
vortex	91.2	8.3	0.5	61.6	238.6

Table 3. Application Time Distribution. This table gives testing and training times for the experimental workloads, and a breakdown of training time among the application, shared libraries, and the operating system kernel. The time breakdown is based on profile samples from the benchmark training run. The execution time is measured using the system clock, which has 17 ms resolution. The time distributions are computed from the profile samples; e.g. 0.1% of the profile samples for *go* were PCs values from the kernel.

We used a suite of nine workloads for this study. Table 1 gives a description of each workload, and Table 2 describes the training and testing input sets we used. Table 3 gives summary statistics for the workloads, including both execution times for the test inputs and total execution time for all training inputs. All experiments for this paper were run in single-user mode with a warm file system cache. Two factors limited our choice of workloads. First, any potential benchmark application had to be compiled with SUIF. As a research compiler, SUIF does not support the full set of language idioms supported by *gcc* or commercial C compilers, and makes less efficient use of CPU time and memory during compilation. As a result we were unable to compile many popular public domain UNIX applications. We also excluded applications that are I/O bound or spend most of their time in the kernel, as they will not benefit significantly from re-optimization.

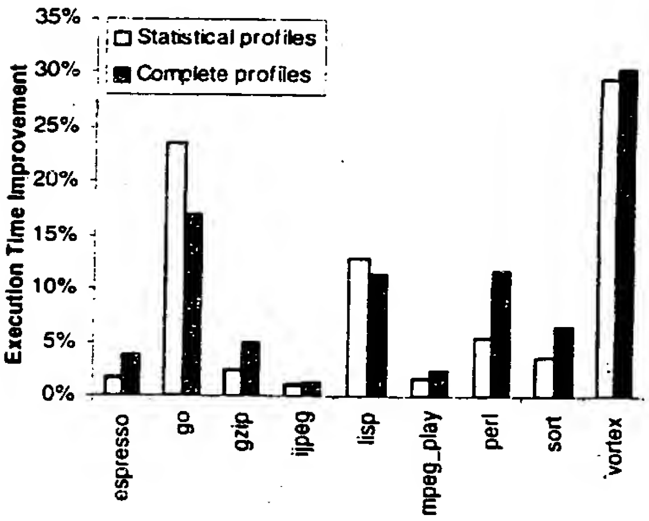


Figure 3. Optimization Results: This figure shows the reduction in execution time for programs optimized by Morph as a percentage of the execution time of the original program. All execution times are the average of ten runs. The improvement in execution time for complete profiles is included for reference.

4.2 Optimization Results

Figure 3 shows optimization results for the nine test workloads. Optimization using profiles gathered by the Morph system improves application performance by up to 27% depending on the application. The results labeled "statistical profiles" in Figure 3 were generated by combining profile samples from the training runs of a workload, generating an optimized executable using the Morph Editor, and then measuring the execution time of the optimized executable on the "test" input data set. Our experience with Morph suggest that minimal training is needed to obtain most of the benefit of optimization. For a detailed discussion of our findings, see [GWZ97]. The execution time improvements given in Figure 3 are computed from the average of ten runs, with execution times measured using the Alpha cycle counter. The standard deviation for all workloads was less than 0.3% with the exception of *go* (1.8%).

To compare performance improvements for automatic optimization in Morph to conventional optimization techniques using complete profiles, Figure 3 also includes a set of optimization results using complete profiles from the execution of instrumented workloads. Overall, the optimization results show that the potential benefit of profile-based optimization is significant and that automatic profiling and optimization with Morph can effectively achieve those benefits.

4.3 Overhead

We classify overhead in Morph according to the three steps in automatic optimization: on-line profile collection, off-line profile processing, and optimization. In this section, we evaluate the overhead of profile collection and profile processing. We do not discuss the overhead of the SUIF optimizations performed by the Morph Editor. SUIF is a research compiler, and is designed for flexibility rather than short compilation times. Ebcioglu and Altman [EA97] describe a dynamic compilation system designed specifically for compilation speed while still performing machine-specific optimizations such as global instruction scheduling. They report that their compiler requires 15 times fewer instructions per generated instruction than *gcc*. Similar techniques could be applied to the Morph Editor to make the cost of optimization low.

4.3.1 On-line Profiling Overhead

Overhead from on-line profile collection comes from two sources: the time to record a PC sample into the in-memory profile buffer and the time required to periodically flush the profile buffer to disk. A problem that arises when attempting to measure overheads on the order of 0.1% for realistic workloads is that the overhead is smaller than the variation that can occur due to non-determinism in the system. A major source of such non-determinism is virtual-to-physical page mapping [KH92]. The standard Digital UNIX kernel uses a bin-hopping policy for allocation of physical memory pages. Although this policy has some desirable performance characteristics [KH92], it has the undesirable property of poor repeatability. For our overhead measurement experiments, we replaced the bin-hopping policy with page coloring [TDF90], a deterministic policy, to improve the repeatability of our experiments. All results reported in this section use the page coloring policy. The problems and pitfalls of page mapping algorithms are well documented in the research literature [KH92, CB93, BCL94].

Table 4 gives the overhead of the on-line component of profiling for our test workloads. To distinguish the impact of profiling from memory system effects, we compare execution times for the Morph kernel with profiling and the Morph kernel with profile collection disabled. Disabling profile collection eliminates most of the instruction overhead for the Monitor and prevents profile sample generation, while preserving code displacements in kernel memory. We also present execution times for a standard Digital UNIX 4.0 kernel. Note that the standard kernel can give slightly better or slightly worse performance than the Morph kernel with profiling disabled. This is an indication of the impact of moving code within the kernel on execution time for the test workloads.

Program	*DU*	*Morph-off*	*Morph*	*Overhead*
espresso	12.051 (0.001)	12.133 (0.001)	12.155 (0.003)	0.18%
go	279.724 (0.193)	282.045 (0.234)	282.498 (0.170)	0.16%
gzip	25.275 (0.010)	24.953 (0.009)	24.981 (0.007)	0.11%
ijpeg	20.770 (0.005)	20.740 (0.004)	20.770 (0.006)	0.14%
lisp	19.319 (0.001)	19.296 (0.001)	19.322 (0.004)	0.13%
mpeg_play	19.262 (0.001)	19.309 (0.001)	19.341 (0.004)	0.17%
perl	46.977 (0.043)	46.253 (0.017)	46.357 (0.020)	0.22%
sort	13.591 (0.010)	13.614 (0.009)	13.624 (0.010)	0.07%
vortex	59.324 (0.006)	59.840 (0.014)	59.982 (0.015)	0.24%

Table 4. On-line overhead for Morph continuous profile collection. This table shows execution times and profiling overhead for our workloads using test inputs and three different systems, the base Digital UNIX kernel (DU), the Morph kernel with profiling disabled (Morph-off), and the Morph kernel with profiling enabled (Morph). All systems used page-coloring for physical page selection. Times are the average of ten runs, in seconds, and were measured using the Alpha cycle counter. All measurements in this table are for a warm file system cache. Standard deviations are given below times (in parentheses). Overhead is computed by comparing execution time with Morph monitoring (Morph) to execution time using a Morph kernel with profiling disabled (Morph-off).

Program	*DU*	*Morph-off*	*Morph*	*Overhead*
strawman	86.38 (0.00)	86.38 (0.00)	86.45 (0.00)	0.07%
strawman 512KB	67.97 (0.01)	68.38 (0.00)	68.73 (0.01)	0.50%
strawman 1MB	82.11 (0.00)	82.17 (0.00)	82.31 (0.01)	0.17%

Table 5. On-line overhead for strawman tests. This table shows execution times and profiling overhead for the three strawman tests. The experimental operating systems are the same as for Table 4. Note that the number of iterations performed by these tests is not the same. Strawman-1MB has a very high CPI (about 18) making it impractical and unnecessary to run a longer test.

To quantify the overhead of profiling, we compare benchmark execution times for the Morph kernel with and without profiling. Table 4 shows that the overhead of profiling in Morph is very small, ranging from 0.07% to 0.24%. Although the absolute overhead is small, the relative difference between the highest (*vortex*) and lowest (*sort*) overheads is large. Referring to Table 1, note that small overheads occur for programs with small text segments, and that larger overheads occur for the programs with large text segments. Programs with small code working sets have relatively little chance of causing instruction cache conflicts with the Morph profiling code. For larger programs, it is more likely that a cache conflict between the application and the kernel profiling code will occur.

To isolate these memory system effects and quantify the relationship between code working set and profiling overhead, we constructed a set of artificial tests, *strawman*, *strawman 512KB*, and *strawman 1MB*. Strawman is an eight-line C program consisting of a tight loop that increments an integer variable. The key features of this program are that it is compute bound with no I/O and no system calls, generates profile samples at the maximum rate, and makes minimal use of the memory system, requiring one line in the instruction cache and no data cache references. The trivial memory reference pattern of the strawman benchmark and the lack of system or library calls make it possible to isolate the CPU overhead of profiling. The results for the strawman test show that the lower bound for profiling overhead when samples are generated at the peak rate is about 0.07% (Table 5).

Programs with larger working sets will tend to cause more cache conflicts with the kernel profiling code. The worst case occurs for a program that occupies the entire second-level cache but does not generate any self-conflict misses. For such a program, no misses occur when there is no kernel interference, but with kernel interference up to two misses can occur for each cache line of kernel code accessed, one to bring kernel code into the cache, and one to restore the displaced user code. We constructed the strawman 512KB test to create this worst-case behavior. The profiling overhead for strawman 512KB is 0.50%, and this gives an estimate of worst-case overhead for our monitor. Our strawman 512KB test is not the absolute worst case for profiling overhead as it only interferes with the kernel in the instruction cache. An artificial test that interferes in a pessimal way for both instruction and data references is conceivable, but variability in kernel and user data layout makes it difficult to construct.

For large programs with self-conflict misses, kernel profiling activity will cause additional instruction cache misses for kernel references but will tend not to cause new user instruction cache misses. An extreme case for our direct-mapped 512KB cache is a program that loops over a 1MB block of code, such that each user instruction reference that crosses a cache line boundary generates a cache miss. We constructed the strawman 1MB test to create this behavior. As expected, the profiling overhead for this case is 0.17%, less than the overhead for the strawman 512KB case.

To gain further intuition into sources of overhead for on-line monitoring, consider the two required activities. The first is the recording of PC samples during the clock interrupt routine. The Morph Monitor adds 72 instructions to each clock interrupt to record a PC sample in the sample buffer, or a total of approximately 74,000 instructions per second. The overhead of these instructions depends on their CPI. For example, for a CPI of three, the overhead would be about 222,000 cycles, or approximately 0.10% for a 233 MHz machine. This is consistent with the overhead seen for strawman and for sort, the smallest benchmark. For applications with a large working set, the CPI of the profiling instructions will be higher. For a CPI of 10, the overhead estimate would be about 740,000 cycles, or approximately 0.32% of the machine. This correlates well with the overheads observed for the test workloads.

The other source of on-line sampling overhead is the writing of profile samples from the in-memory sample buffer to disk. This overhead depends on the amount of idle time, which does not generate samples, and on the frequency of logged events such as context switches and *exec()* system calls. With a 1024 Hz clock interrupt rate, profile samples are generated at a peak rate of approximately 8 KB per second. With our current monitor we have observed that 1-24% of the unprocessed profile information is for logged events, with the remainder occupied by profile samples. On days when the system is relatively busy and a large number of profile samples are generated, logged events occupy less than 12% of the total space.

The Monitor flushes the sample buffer to disk every 10 seconds. Assuming conservatively that the profile samples are generated at the peak rate of 8 KB per second and that the buffer contains 75% profile samples and 25% logged events, the writing of the profile buffer to disk will generate 110 KB of disk write operations every ten seconds. The samples are copied once on their way to disk, and this copy activity dominates the CPU activity required for the write. This generates 110 KB of copy activity for each ten seconds interval in our pessimistic scenario. Assuming 8-byte writes and a pessimistic ten cycles per write, this would add approximately 140,000 cycles of memory latency to each ten seconds of computation, or 0.006% overhead. The logging of profile samples to disk also consumes some amount of disk bandwidth, although at 11 KB per second the bandwidth is insignificant when compared to the write bandwidth of a modern high-performance disk. As the writes are contiguous and the frequency of writes is low, the seek overhead required by writes should not be significant, except for a badly fragmented disk.

Overall, these overhead estimates are consistent with the on-line profiling overheads observed for our test workloads. We conclude that profiling overhead in Morph is very small and has a negligible impact on overall performance.

4.3.2 Off-line Profile Processing Overhead

Processing of raw PC samples occurs off-line and is deferred until off-peak hours to avoid interfering with interactive users. For our current version of the Manager, profile samples can be processed at a rate of 60 MB per minute. Again using the pessimistic assumption of 75% profile samples and 25% logging entries, profile samples are produced at a peak rate of about 640 KB per minute. Given that there are 1,440 minutes in a day, 900 MB (uncompressed) is an upper bound on the amount of profile samples and logging information produced per day. It is not necessary to provide this much staging space for profile samples. In our experience, the amount of profile samples generated in a day is typically tens of megabytes, and profile staging space can be limited to this level. To control the amount of disk space required, the system can simply discard samples. For systems with sufficient idle time, use of a low-priority daemon to post-process the samples more frequently can also reduce staging space requirements.

Long-term profile storage occurs in the form of basic block counts. After being computed from the raw samples, the basic block profiles of individual programs are integrated into the intermediate form representation of the program. Table 6 lists the disk space occupied by long-term storage of basic block profiles for our test application suite. The sizes given are for uncompressed profiles. Compression further would reduce profile storage requirements. The profile size is roughly proportional to the size of the program code segment. As the number of basic blocks for a program is stable, the profile sizes do not change over time. For our benchmark suite, the space requirements for storage of basic block profiles ranges from 29% to 82% of the program text size.

Program	*Executable*	*Text*	*Profile*	%
sort	80	56	46	82%
gzip	152	96	49	51%
lisp	216	112	94	84%
mpeg_play	328	248	73	29%
espresso	376	248	124	50%
ijpeg	400	248	121	49%
go	608	416	225	54%
perl	664	488	367	75%
vortex	1280	800	440	55%

Table 6. Long-term Profile Storage. This table shows executable, text, and profile sizes in kilobytes for our test workloads. The right-most column shows profile size as a percentage of text segment size.

The space requirement for long-term profile storage will tend to increase in more sophisticated versions of Morph. Heuristics to decide when to re-optimize a program will require history information to detect when changes in program usage patterns justify re-optimization. Also, although our current set of optimizations is effective with basic block profiles, new optimizations or refinements to our current optimizations may require different types of profiles. Further research will be required to balance the tradeoff between profile storage space and effective re-optimization.

5. Discussion

Our work to date has demonstrated that automatic profiling and optimization is both efficient and effective. There are a number of issues that must be resolved to make the system practical for more widespread use.

Our current system uses trivial heuristics to decide when to re-optimize (i.e. re-optimize when triggered manually, re-optimize when new profiles are available). We suspect that more sophisticated heuristics (i.e. re-optimize when program usage patterns change) could lead to substantial improvements in the system. These heuristics must achieve a balance between optimizing rarely, so that re-optimization occurs promptly and within the budget of available idle CPU cycles, and optimizing frequently, to maximize the benefit from profile information. In our continuing work we are developing the profile analysis techniques needed to implement improved heuristics.

Our current system uses simple methods for combining profile information. The continuous generation of profiling data raises the issue of how to age profile data as it is collected over an extended period of time. There are a number of simple approaches. This is a part of our continuing work.

Before automatic optimization can be incorporated into mainstream computing platforms, an infrastructure must be put into place that allows intermediate form representations of program modules to be recovered from the application distribution. This issue must be fully resolved before Morph-style automatic optimization becomes widely used. From a purely technical point of view, the most straightforward solution would be to augment the shipping versions of executables with an intermediate form representation of the program. This solution has major practical drawbacks, however, as it requires new software standards and standards compliance across the software industry. Another possibility is to develop new ways of acquiring the additional information needed to derive intermediate form program representations from current executables. Digital's FX!32 [CH97, Rub96] provides an example of progress in this approach. Our PostMorph tool represents another alternative.

One of the greatest barriers to obtaining conclusive results for Morph is the lack of a compelling suite of interactive benchmarks. The ideal test for our system would be a suite of large, interactive benchmarks in a format that permits re-optimization. Unfortunately, source or intermediate form representations for such benchmarks are not commonly available. Recent developments in late-code optimization [CL96, Rub96, RVL97] offer some promise in addressing these issues.

6. Conclusions

This paper describes a system for continuous low-overhead profile collection designed to meet the unique requirements of automatic optimization. We achieve these low overheads through statistical sampling of all system activity and by deferring processing of profile samples for off-line processing. Our results show that continuous profiling can be achieved with very low overhead and that the resulting profiles are effective when applied in profile-based optimization. As high performance processors become increasingly dependent on super-scalar issue and deep memory hierarchies for sustained performance, the importance of these optimizations will increase.

The prospect of practical operating system support for automatic optimization introduces a host of opportunities for new research, both in systems and in compilation. Our Morph testbed system demonstrates how a practical framework for automatic optimization can be implemented for a UNIX system, and provides a foundation for further research in this area.

Acknowledgements

Many members of the Harvard Machine SUIF project contributed time and brainpower in maintaining the machine-specific code optimizations used in this paper. We would like to specifically acknowledge Gang Chen and Cliff Young, who provided support for the optimizations in the Morph Editor.

we would also like to thank the anonymous reviewers for their useful comments. We would especially like to thank our shepherd Jeff Mogul who provided valuable feedback during the last stages of the preparation of this paper.

The Morph project is supported by gifts from Intel and Digital Equipment Corporation. Brad Chen is supported in part by an NSF CAREER Award, CCR-9501365. Mike Smith is supported in part by DARPA grant number NDA904-97-C-0225 and by a National Science Foundation Young Investigator Award, grant number CCR-9457779.

Bibliography

[ABD97] J. Anderson. L. M. Berc. J. Dean. S. Ghemawat. M. R. Henzinger. S. Leung. R. L. Sites. M. T. Vandevoorde. C. A. Waldspurger. and W. E. Weihl. "Continuous Profiling: Where Have All the Cycles Gone?" In *Proceedings of the 16th ACM Symposium of Operating Systems Principles* (in this volume). October 1997. See also http://www.research.digital.com/SRC/dcpi/

[BCL94] B. Bershad. J. B. Chen. D. Lee. and T. Romer. "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches." In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. pages 158-170. October 1994.

[CB93] J. B. Chen and B. Bershad. "The Impact of Operating System Structure on Memory System Performance." In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM. pages 120-133. December 1993.

[CH97] A. Chernoff and R. Hookway. "DIGITAL FX!32 - Running 32-Bit x86 Applications on Alpha NT." To appear in *Proceedings of the USENIX Windows NT Workshop*. USENIX Association. Berkeley CA. August 1997.

[Chr96] P. Christy. "IA-64 and Merced—What and Why." In *Microprocessor Report*. pages 17-19. 30 December 1996.

[CL96] R. Cohn and G. Lowney. "Hot Cold Optimization of Large Windows NT Applications." In *Proceedings of the 29th Annual International Symposium on Microarchitecture*. IEEE. pages 80-89. December 1996.

[CMH96] T. Conte. K. Menezes. and M. A. Hirsch. "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer." In *Proceedings of the 29th Annual International Symposium on Microarchitecture*. IEEE. pages 201-211. December 1996.

[Col95] Colusa Software. "Omniware. A Universal Substrate for Mobile Code." White paper from Colusa Software. http://www.colusa.com/

[EA97] K. Ebcioglu and E. Altman. "DAISY: Dynamic Compilation for 100% Architectural Compatibility." In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM. pages 26-37. June 1997.

[Fis97] J. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction." In *IEEE Transactions on Computers*. Volume 30. Number 7. pages 478-490. July 1981.

[FF92] J. Fisher and S. Freudenberger. "Predicting Conditional Branches from Previous Runs of a Program." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. pages 85-95. October 1992.

[FK96] M. Franz and T. Kistler. "Slim Binaries." Technical Report No. 96-24. Department of Information and Computer Science. University of California. Irvine. June 1996.

[Goo97] D. Goodwin. "Interprocedural Dataflow Analysis in an Executable Optimizer." In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. pages 122-133. November 1997.

[Gwe96] L. Gwennap. "Bringing Parallelism Out of the Closet." In *Microprocessor Report*. pages 14-15. 9 December 1996.

[GWZ97] N. Gloy. Z. Wang. X. Zhang. J. B. Chen. and M. D. Smith. "Optimization with Statistical Profiles." Technical Report TR-02-97. Center for Reliable Computing. Division of Engineering and Applied Sciences. Harvard University. April 1997.

[KH92] R. Kessler and M. Hill. "Page Placement Algorithms for Large Real-Index Caches." In *ACM Transactions on Computer Systems*. Volume 10. Number 4. pages:338-359. November 1992.

[LRW91] M. Lam. E. Rothberg. and M. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. pages 63-74. April 1991.

[LS95] J. Larus and E. Schnarr. "EEL: Machine Independent Executable." In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*. ACM. pages 291-300. June 1995.

[OSF93] Open Software Foundation Research Institute. *ANDF Collected Papers. Volume IV*. Open Software Foundation. December 1993. See also http://www.opengroup.org/RI/andf/

[PH90] K. Pettis and R. Hansen. "Profile Guided Code Positioning." In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. ACM. pages 16-27. June 1990.

[Rub96] N. Rubin. "FX!32." Work-in-progress talk at the *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. October 1996. See also http://www.digital.com/info/semiconductor/amu/fx32/fx.html

[RVL97] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, B. Bershad, H. Levy, and J. B. Chen, "Etch, an Instrumentation and Optimization tool for Win32 Programs." To appear in *Proceedings of the USENIX Windows NT Workshop*, USENIX Association, August 1997. See also http://www.cs.washington.edu/homes/bershad/Etch/

[SE94] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools." In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, ACM, pages 196-205, June 1994. See also Research Report 94/2, Western Research Laboratory, Digital Equipment Corporation.

[Smi91] M. Smith, "Tracing with Pixie." Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, CA, USA, November 1991.

[Smi96] M. Smith, "Extending SUIF for Machine-dependent Optimizations." In *Proceedings of the First SUIF Compiler Workshop*, Stanford, CA, pages 14-25, January 1996.

[SUIF94] Stanford SUIF Compiler Group, "SUIF: A Parallelizing and Optimizing Research Compiler." Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[SW92] A. Srivastava and D. Wall, "A Practical System for Intermodule Code Optimization at Link-Time." In *Journal of Programming Languages*, Volume 1, Number 1, pages 1-18, March 1993. See also Research Report 92/6, Western Research Laboratory, Digital Equipment Corporation.

[TDF90] G. Taylor, P. Davies, and M. Farmwald, "The TLB Slice – A Low-Cost High-Speed Address Translation Mechanism." In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ACM, pages 355-363, 1990.

[Wal92] D. Wall, "Systems for Late Code Modification." In *Code Generation -- Concepts, Tools, Techniques*, Springer-Verlag, pages 275-293, 1992.

[WP87] D. Wall and M. Powell, "The Mahler Experience: Using an Intermediate Language as the Machine Description." In *Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM, April 1987. See also Research Report 87/1, Western Research Laboratory, Digital Equipment Corporation.

[YJK97] C. Young, D. Johnson, D. Karger, and M. Smith, "Near-optimal Intraprocedural Branch Alignment." To appear in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, ACM, pages 183-192, June 1997.

A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization

Matthew C. Merten Andrew R. Trick Christopher N. George John C. Gyllenhaal Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois
Urbana, IL 61801
Email: {merten, atrick, c-george, gyllen, hwu}@crhc.uiuc.edu

PD: 02-05-1999
P: 136-147 (12)

Abstract

This paper presents a novel hardware-based approach for identifying, profiling, and monitoring hot spots in order to support runtime optimization of general-purpose programs. The proposed approach consists of a set of tightly coupled hardware tables and control logic modules that are placed in the retirement stage of a processor pipeline removed from the critical path. The features of the proposed design include rapid detection of program hot spots after changes in execution behavior, runtime-tunable selection criteria for hot spot detection, and negligible overhead during application execution. Experiments using several SPEC95 benchmarks, as well as several large WindowsNT applications, demonstrate the promise of the proposed design.

1 Introduction

Optimizing compilers can gain significant performance benefits by performing code transformations based on a program's runtime profile. Traditionally, profiles are collected by running an instrumented version of the executable. However, because this profiling technique incurs a large overhead, applications are only profiled prior to distribution on a set of sample inputs. Consequently, they cannot be adaptively optimized in order to account for changes in program behavior or to take advantage of variations in the production system. More recently, low-overhead methods of profiling have been developed based on statistical sampling [1] [2] [3] [4]. The basic approach, however, remains the same: profile information for the program's entire execution is averaged into a large database and later fed back into a static compiler. This approach is undesirable for three reasons. 1) The entire profile of each application must be continuously maintained at runtime on the production system. 2) The profile represents only average behavior across an extended period of time. 3) A significant length of time may pass before variations in the program's behavior are detected. This paper addresses these problems by presenting a new method for rapidly, accurately, and transparently collecting profile information with minimal runtime overhead. The proposed hardware approach provides a strong foundation for a runtime optimizing system.

Although static optimizations are essential to application performance, additional opportunities can be exploited by continuously profiling and reoptimizing the code. For instance, many types of optimizations cannot be performed without more specific runtime information. These include optimizing code based on value invariance [5] [6] and inlining dynamically linked library functions [7]. The traditional, static approach also has the disadvantage that aggressive optimization can only be applied selectively, based on average profile weights and other criteria [8] [9]. Optimizations that cause excessive code growth, for example, must be applied conservatively by the static compiler [10]. However, a runtime optimizer can more aggressively apply these optimizations by targeting small regions of the program that represent the critical execution path at a particular time. To support targeted optimization, our runtime profiler extends the traditional role of profilers and captures temporal information as well as reporting the relative importance of basic blocks. The profiler identifies code that can be optimized quickly compared with amount of execution time that will subsequently be spent in code. Additionally, code that yields a significant short-term benefit is given priority since a runtime optimizer may not have enough memory at its disposal to retain all optimized code indefinitely.

We have observed that many applications exhibit behavior conducive to runtime profiling and optimization. For example, program execution often occurs in distinct phases, where each phase consists of a set of code blocks that are executed with a high degree of temporal locality. When a collection of intensively executed blocks also has a small static footprint, it represents a highly favorable opportunity for runtime optimization. We will refer to such sets of blocks and their corresponding periods of execution as ***hot spots***. A runtime optimizer can take advantage of execution phases by isolating a group of hot spots that are active for each phase. Ideally, aggressively optimized code would be deployed early in the phase and the optimized code used until execution shifts to another phase. Optimized hot spots that are no longer active may then be discarded, if necessary, to reclaim memory space for newly optimized code.

Several common types of programs exhibit this execution phase behavior [11], such as compilers, graphics packages, and scripting engines. Because these applications typically implement a wide range of functionality, it is often intractable to aggressively optimize for every possible runtime scenario. Yet, these programs should benefit from runtime optimization because they are divided into focused tasks, or phases of execution. Compilers, for example, allow the user to select a variety of optimizations and other behavior at runtime, and they will often perform multiple passes across the input, exercising different functionality each time. Graphics packages also allow a wide variety of tasks to be selected at runtime, each of which may perform an intensive transformation on the data. Furthermore, it is likely that these transformations can be highly tuned when the data set is available during optimization. A similar situation exists in many scripting engines because a sizeable number of high-level commands are available that perform intensive operations on the data. Yet, for any single script, only a small selection of these routines may be utilized.

A concrete example of this behavior can be seen in 134.perl running the jumble training input from the SPEC95 benchmark suite. As shown in Figure 1, this benchmark contains three primary, distinct phases of execution with one hot spot per phase. Hot spot 1 runs for 72 million instructions, hot spot 2 for 1.35 billion, and hot spot 3 for 200 million. The first hot spot consists of reading in a dictionary and storing it in an internal data structure. The second hot spot processes each word in the dictionary, and the third scrambles a selected set of words in the dictionary.

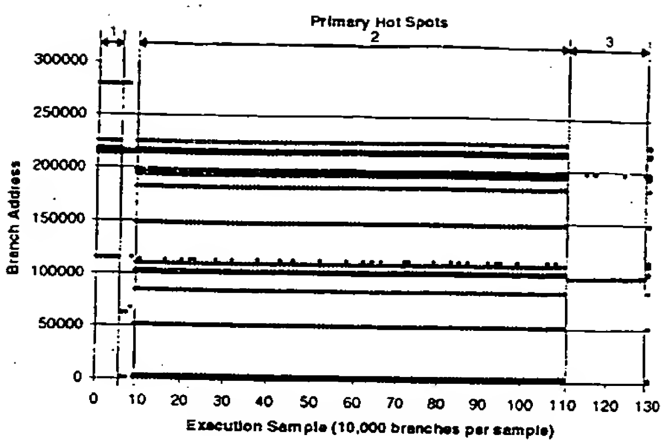


Figure 1. Important branches executed in each execution sample for 134.perl. Each data point represents a branch that executed at least 40 times within the sample duration of 10,000 branches. The sample period is 2,000,000 branches.

The second of these hot spots serves as an excellent example of why runtime optimization is needed. The input script causes perl to call *split* which breaks up an input word into individual letters, then to call *sort* which sorts those letters. The first function, *split*, calls a complicated regular expression matcher with an empty regular expression. This region of code would benefit from partial inlining and code layout, followed by classical optimization of the few exercised blocks in the regular expression matcher and split. A static compiler could perform this optimization, but the larger code size and compile time would not benefit most input scripts. The second function, *sort*, calls the library function *qsort* which then calls a perl-specific comparison function. Less than half of the code in the comparison function is ever executed because only single characters are actually sorted. This is another example where inlining is important because of the very frequent calls to the comparison function. However, a link time optimizer or runtime optimizer is needed to support inlining across library and application boundaries. Figure 2 shows a branch profile for a typical 10,000 branch sample of this second hot spot and clearly demonstrates that a small number of static branches account for the vast majority of the dynamic instances in the sample.

Our strategy for detecting runtime optimization opportunities consists of two stages. Initially, our profiler must detect hot spots as they emerge during execution. To do this, we track the execution frequency of code blocks across a relatively short period of time. When a set of blocks fit the characteristics of a hot spot, the profiler notifies the runtime optimizer. During the detection stage, arc weights are also accumulated for the cache. Y is accessed three times followed by a long stream of X's. If X is not replaced in the cache by Y, then three I-cache misses will result, one for each access to Y. An LRU policy predicts that the next access will be like the previous one and will replace X with Y in order to save a miss. After the third Y, X will again be accessed, resulting in another miss. In this case, the LRU policy was able to reduce three misses down to two but the replacement of X was required.

The BBB, however, must accurately record the most frequently executed blocks and their profiles rather than the most recently accessed. Therefore, implementing an LRU replacement policy and allowing a rare block to replace a frequently executed block is unacceptable. Instead, branches that conflict with existing BBB entries are simply discarded. The function of branch replacement is controlled by the aforementioned refresh timer. A more serious conflict occurs when two branches are important and map into the same cache location. As long as these conflicts are relatively rare, the runtime optimizer can still recover by inferring the profile value of the missing branches. Using Kirchhoff's First Law and other heuristics [12], the input and output weights of the block can be estimated based on the branches that were accurately profiled.

While the I-cache and BTB collect information about important instructions and branches, respectively, it is infeasible to combine the BBB with either of these two structures. In addition to the previously discussed replacement policy differences, the BTB and I-cache are both potentially on the critical path, and adding complexity in these two structures may adversely affect cycle time. In addition, since only true branch behavior should affect the BBB counters, incorrectly speculated accesses must not be allowed to affect the BBB entries. Although the BBB counters in these structures could be updated after the branches are resolved, the strict timing constraints and complex hardware required to interface with the BTB and I-cache warrant the cost of storing the BBB entries separately. Thus, the BBB is best implemented as a stand-alone structure in the retirement phase because of its simplicity and inherent accuracy.

2.1.2 Hot Spot Detection Counter

Once candidate branches have been identified in the Branch Behavior Buffer, they must be monitored to determine whether the corresponding blocks may be considered a hot spot and, thus, useful candidates for optimization. We have found two criteria that should be satisfied before a group of candidate blocks is dubbed a hot spot. First, the candidate blocks should be active for a specified minimum amount of time. Second, the candidate branches should account for at least a certain percentage of the total branches executed during this time. We define the minimum percentage over the time interval to be the *threshold execution percentage* and the actual dynamic percentage over the interval to be the *candidate execution percentage*.

In order to minimize disruption of the system during hot spot detection, we perform the detection in hardware using a *Hot Spot Detection Counter* (IIDC), shown in Figure 3. The Hot Spot Detection Counter is an up/down counter used to detect when the set of candidate branches reaches the threshold execution percentage. The counter is implemented as a saturating adder that is initialized to the maximum value. It counts down by D for each candidate branch executed or counts up by I for each non-candidate branch executed, where the determination of D and I will be discussed later. When the candidate execution percentage exceeds the threshold percentage, the counter begins to move down. If the candidate execution percentage remains higher than the threshold for a long enough period of time, the counter will decrement to zero. At this point, the operating system will be triggered either via an interrupt or by setting a flag that is checked the next time the operating system is invoked.

The difference between the candidate execution percentage and the threshold execution percentage determines the rate at which the counter decrements (i.e., the rate at which the hardware identifies the hot spot). This corresponds to our observation that hot spots become more desirable as they either account for a larger percentage of total execution or run for a longer period of time. It is assumed that hot spots which have been active over a longer period of time are less likely to be spurious in their execution and are more likely to continue to run after optimization has been complete.

Our experiments have shown this approach to work quite well and to detect all of the major hot spots in our benchmarks. There are three primary scenarios where there is no hot spot to be found, and thus the HDC will never reach zero:

1. Few branches execute with sufficient frequency to be marked as candidates, and collectively, they do not constitute a large percentage of the total execution. Thus, even if they were classified as a hot spot and optimized, only a small benefit is likely.

2. The number of branches that execute frequently enough to be considered candidates is too large to fit into the BBB. If the branches that are able to enter the BBB do not account for a large enough

percentage of execution, they will not be identified as a hot spot. This may happen if the execution profile of the region is very flat. Although some benefit may be gained by optimizing all the frequent branches, the overhead of optimizing such a large region would most likely be prohibitive.

3. The execution profile is not consistent. In this case, a small set of branches may account for a large percentage of execution over a short time, but the execution shifts to a different region of code before the Hot Spot Detection Counter saturates. Optimizing a region of code that only executes spuriously is unlikely to yield much benefit.

As in these three scenarios, branches that collectively do not constitute a hot spot may be locked into the BBB. Since the HDC has not identified these branches as a hot spot, it is possible that the execution has shifted to a new, potentially more important region of code or is not an attractive optimization opportunity. Therefore the BBB will be periodically purged by the *reset timer* to make room for new branches. This timer is similar to the refresh timer BBB but clears all entries in the table, rather than only the non-candidate branches. The reset interval should be large enough to allow the HDC to saturate on valid hot spots but small enough to allow quick identification of a new phase of execution.

Once the threshold execution percentage X_t required for a hot spot has been selected, the HDC increment and decrement values should be chosen. D is the decrement value when a candidate branch is encountered (*candidate hit*), and I is the increment value for a branch that is not in the table or is not yet marked as a candidate (*candidate miss*). Let X be the actual candidate execution percentage. For a given D and I, the counter will decrease when the candidate execution percentage multiplied by the decrement value is greater than the percentage of non-candidates multiplied by the increment value. This is represented by the equation:

$$X * (-D) + (1 - X) * (I) \leq 0 \tag{2}$$

Rearranging the terms and solving for X yields the formula for minimum percentage:

$$X \geq \frac{I}{D + I} = X_t \tag{3}$$

Equation 3 shows that the counter decreases when the percentage of execution is above the threshold, as determined by I and D.

Given the increment and decrement values, the size of the HDC can be chosen to achieve a minimum detection latency. Let N be the minimum number of branches executed before a hot spot is detected. For detection to occur, the following inequality must hold:

$$N * X * (-D) + N * (1 - X) * (I) \leq -HDC\_MAX\_VAL \tag{4}$$

The latency for detecting a hot spot is determined by the following equation:

$$N = \frac{HDC\_MAX\_VAL}{(D + I) * (X - X_t)} \tag{5}$$

As the candidate execution percentage further surpasses the threshold, the detection latency decreases. The latency can also be decreased independently of the candidate execution percentage by increasing I and D such that X_t remains constant.

2.2 Monitor Table

The purpose of the *Monitor Table* is to determine when hot spot profiling is necessary. This hardware mechanism is continuously running, watching program execution and comparing the executing branches to those already determined to be in hot spots. When the program is executing in the known hot spots, the system is said to be in *monitor mode*, which is the steady-state mode of execution. The system enters *profile mode*, and the BBB is enabled when the Monitor Table determines that execution has strayed from the known set of hot spots. Note that the Monitor Table continues to operate during profile mode, watching for execution to return to the set of known hot spots. If this situation were to occur, the BBB would be deactivated, since it is unnecessary and costly to extract and possibly optimize a hot spot that has already been processed. If a new hot spot is found, the operating system is notified which then extracts the hot spot from the BBB; the new hot spot is entered into the Monitor Table; and the BBB is deactivated. When new, optimized code is ready for deployment, the branches in the new code will be added to the table just prior to deployment to avoid hot spot detection in the optimized code.

In order for the system to detect when execution strays from the known set of hot spots, the hardware must be aware of those hot spots which have already been identified. In an ideal Monitor Table, the addresses of all branches in all known hot spots would be placed into a tag array as shown in Figure 5. When a branch is executed, the instruction address is looked up in the tag array. Its presence in the array indicates execution in a hot spot. An up/down counter called the *Monitor Counter* is used to track long-term execution trends and operates much like the HDC. It counts

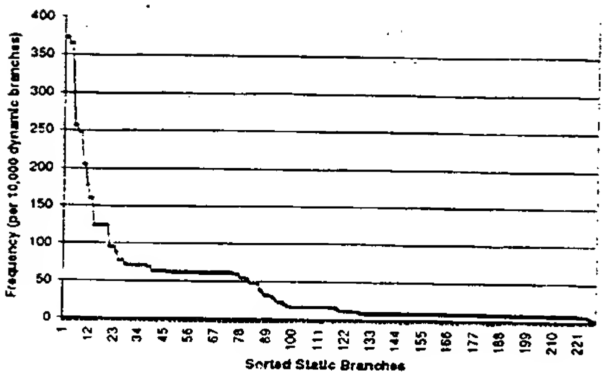


Figure 2. Profile distribution for the second primary hot spot in 134.perl. Branches sorted from most to least frequent.

hot spot to aid profile-driven optimizations. Once a hot spot has been detected, we add the hot spot to a monitor table and disable the profiling hardware. During the second stage, we monitor the program and compare the executing code against the table of currently active hot spots. When execution strays from the monitored hot spots, the profiler is enabled to allow detection of new hot spots.

The rest of this paper is organized as follows: Section 2 explores the mechanisms for performing hot spot detection during runtime; Section 3 details our experimental environment and results; Section 4 examines research related to our endeavors; and Section 5 summarizes our efforts and future directions.

2 System Components

Our proposed hot spot detection scheme uses three criteria to classify a region of code as a hot spot. First, the region must have a small static code size to facilitate rapid optimization. Second, the hot spot must be active over a certain minimum time interval so that it is likely to have an opportunity to benefit from runtime optimization. Finally, the instructions in the selected region of code must account for a large majority of the dynamic execution during its active time interval. These three criteria are sufficient to detect code regions that can benefit most from runtime optimization without placing unnecessary restrictions on the type of hot spots that can be identified.

Our proposed scheme must both detect when the execution is in a hot spot and also gather profile data for that hot spot. When a valid hot spot has been discovered, the operating system is notified so that it can invoke the runtime optimizer. Additional hardware is used to ensure that further notifications occur only when significant new hot spots are detected. Our proposed implementation consists of the following components:

Hot Spot Detector Hardware that collects control flow profiles and identifies a collection of important blocks that comprise a hot spot. The Hot Spot Detector contains a Branch Behavior Buffer (BBB) to store the branches and profile data and a Hot Spot Detection Counter (HDC) to track the percentage of dynamic execution accounted for by the hot branches.

Monitor Table Hardware that maintains a collection of previously discovered hot spots that the runtime optimizer has already examined. It monitors program execution, noting when execution strays from previously detected hot spots so that profiling can be restarted.

Operating System Support Software that reads the hot spot information from the BBB and adds the hot spot blocks to the Monitor Table. It can also assemble the blocks into a region in order to call the runtime optimizer.

The Hot Spot Detector and Monitor Table can easily tolerate a rather large latency when recording information about program execution. Therefore, our proposed hardware can be deeply pipelined and located off the critical path so that it does not affect processor performance. While invoking the operating system does incur a penalty, our experimental data shows that the number of operating system interrupts is insignificant relative to the total execution time. This is because the operating system is typically invoked only when a new optimization opportunity is detected.

The following subsections describe each of the three main components in the context of a single process system. This section concludes with a description of the extensions necessary for a multiprocess system and other enhancements.

2.1 Hot Spot Detector

The first step in the process of identifying hot spots is to detect the frequently executed blocks. By employing a hardware scheme, operating system overhead can be eliminated from this portion of the process and detection can be completely transparent to the system. Such blocks can be easily collected in hardware by gathering the branches that define their boundaries. By examining branch execution and direction information, a control flow graph with arc weights can be constructed. Through this hardware scheme, reasonably accurate

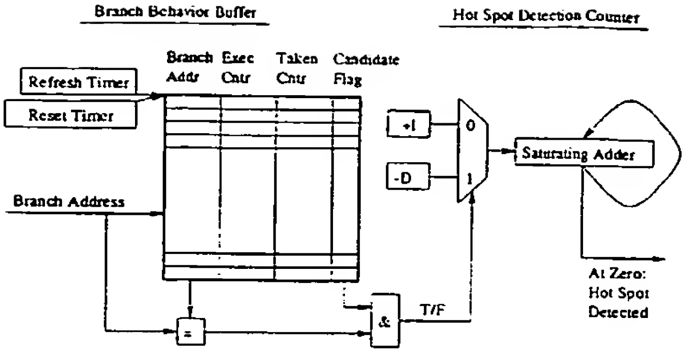


Figure 3. Hot Spot Detector hardware.

profile information will be collected which can benefit the runtime optimization effort.

2.1.1 Branch Behavior Buffer

To achieve these collection goals in hardware, a cache structure reminiscent of a *Branch Target Buffer* (BTB) will be used. As depicted in Figure 3, the structure, called a *Branch Behavior Buffer* (BBB), will be indexed on branch address and will contain several fields: tag (or branch address), branch execution count, branch taken count, and branch candidate flag. When the processor retires a branch instruction, the instruction address will provide the index into the BBB. Each time a branch address is found in the BBB, its execution counter is incremented. If the branch is taken, the taken counter is also incremented. The combination of these two values constitutes an arc weight profile of the recently executed code. Note, it is possible for a branch to execute more times than can be represented by the execution counter. When the execution counter reaches its maximum value, the hardware stops incrementing both the execution and taken counters to preserve their ratio. As long as the number of branches that reach saturation is small, the profile will still accurately represent the relative importance of the branches.

The purpose of the BBB is to collect and profile only frequently executed branches whose corresponding blocks account for a vast majority of the dynamically executing instructions. Branches that execute frequently during profiling may be part of a potential hot spot, and we refer to such branches as *candidate branches*. In order to make this determination, an executed branch is temporarily allocated an entry in the BBB and monitored over a short interval. If the execution of that branch is frequent enough during that interval, its execution counter will surpass a predefined *candidate threshold*. The candidate threshold is associated with a single bit in the execution counter, as shown in Figure 4. When this bit is triggered, the candidate flag is set, and the entry remains marked as a candidate while the execution counter continues to collect profile data. Thus, the BBB only needs to be large enough to hold the candidate branches and potential candidate branches for a given interval.

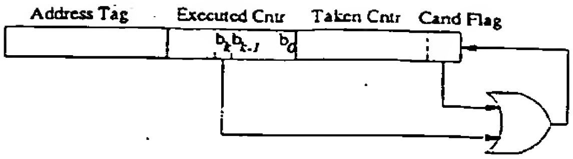


Figure 4. Fields in each Branch Behavior Buffer entry.

To ensure that only frequently executing branches are marked as candidates, a *refresh timer* periodically clears entries from the BBB that have not surpassed the candidate threshold. The refresh timer is simply a global counter that increments each time a branch instruction is executed and triggers a refresh of the BBB when it reaches a certain value. Refreshing the BBB flushes the insignificant entries and ensures that each branch marked as a candidate accounts for at least a minimum percentage of the total dynamic branches during a fixed interval. The minimum percentage of execution required of candidate branches can be expressed as a *candidate ratio*. Thus,

$$CANDIDATE\_RATIO = 2^k/2^n, \qquad (1)$$

given that the size of the refresh timer is n bits, and the candidate threshold is represented by bit b_k of the execution counter.

To accurately represent a hot spot, the BBB must be able to allocate entries for most of the branches that are important in the hot spot. The BBB size should be equal to or larger than the total number of branches present within a hot spot. If the BBB is too small, then insignificant branches may prevent important branches from entering during the first few refresh intervals. Statistically, the important branches will eventually get entries in subsequent refresh intervals, but the profile accuracy may be somewhat compromised and the reporting of hot spots may be delayed.

Another problem is the possibility of indexing conflicts that are inherent in any cache structure. As has been seen in the BTB or I-cache, making the structures set associative eliminates many of the indexing conflicts.

Although organized like a set associative cache, the BBB's behavior differs in terms of replacement policy. The BTB and I-cache often use a least recently used (LRU) policy in order to reduce cache misses. For example, suppose X and Y are instructions that map to the same I-cache location, and X is currently in the

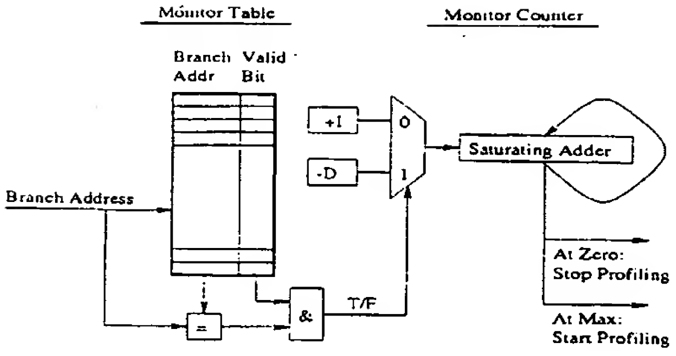


Figure 5. Monitor Table hardware.

down when a hot spot branch is executed and counts up when a non-hot spot branch is executed. When the Monitor Counter saturates at the maximum value in monitor mode, a high percentage of recent branches outside of the known hot spots have been executed, indicating a possible transition to a new hot spot. At this time, profile mode is resumed. Similarly, when the system is in profile mode and the Monitor Counter reaches zero, program execution must have returned to the set of known hot spots. At this time, the BBB is deactivated, and monitor mode is resumed. When monitor mode is entered from profile mode, the Monitor Counter is initialized to zero, indicating that execution is in a hot spot. This is a valid assumption because either the Monitor Counter just saturated at zero to end profile mode, or a hot spot was just detected and execution is likely to continue in the hot spot.

As in the HDC, the increment and decrement values for the Monitor Counter determine the threshold ratio of hot spot to non-hot spot branches. Although a minimum ratio of hot spot to non-hot spot branches must be maintained to remain in monitor mode, this ratio should not be as high as in the HDC. A lower ratio is used for the monitor hardware to allow the behavior of the hot spots to vary slightly without reentering profile mode. Once this ratio is determined, the same formula used for the HDC can be used to derive suitable increment and decrement values for the Monitor Counter.

2.3 Operating System Support

The interface between the BBB and the operating system was designed to be simple: the BBB signals the OS only when a hot spot has been detected, and the operating system copies the BBB table contents into system memory upon hot spot detection. This interface minimizes operating system involvement with hot spot detection. Specifically, cycles are not stolen away by the OS from the user application unless a hot spot is actually detected.

In order for the OS to read the BBB contents, the BBB must be either hardware addressable or readable through special instructions. Once extracted, the branches are stored internally for access by the run-time optimizer. They are also processed and written to the Monitor Table, again via hardware addressing or via special instructions. Since the table is a cache, conflicts over entries in the Monitor Table can arise. In this situation, the OS must either remove an old hot spot that contains a conflicting branch, thus accurately preserving a smaller set of hot spots, or simply eliminate one of the conflicting instructions from the table, thus slightly compromising the recognition of one of the known hot spots. Both of these situations are suboptimal because code previously detected as being part of a hot spot may now trigger the profiling system. The OS is also responsible for deploying optimized code and entering it into the Monitor Table. Furthermore, the operating system has control over several system parameters (BBB increment and decrement values, etc.) and can adjust them based on quality of the hot spots being collected. The actual operating system policies are beyond the scope of this paper.

2.4 Multiprocess support

Thus far, the hardware design has assumed single process execution. Operation becomes slightly more complicated when considering the context switching abilities of microprocessors. It is the responsibility of the operating system to correctly maintain the state of the proposed hardware in a multiprocess environment as it is required to do for traditional hardware components.

Because of the expense of swapping out even a subset of the BBB during context switches, the hot spot detection hardware is designed to operate in single process mode, persisting across context switches. Because the BBB responds to hot spots quickly, it is only active during a small percentage of an application's execution. Because of this low utilization, a single BBB can be shared among multiple processes. The BBB could be configured to search for hot spots associated with a particular process ID, thread ID, or code segment. This ID would be stored in a control register.

Unlike the BBB, the Monitor Table is always in use by each process. Since swapping a table in and out at each context switch would be extremely costly, all processes could share a single table. In order to accomplish this, an up/down Monitor Counter is necessary for each process or active subset to effectively track hot spot behavior. Furthermore, each entry in the Monitor Table must also be tagged with its process ID. This ID serves as a tag for comparison purposes when determining a hit or miss and for determining which Monitor Counter to update.

When a Monitor Counter saturates indicating that profiling is necessary, the BBB must first be allocated to that particular process. A simple check of the BBB process ID control register can be made to determine BBB ownership. If the requesting process owns the BBB, profiling can continue without delay. Otherwise, arbitration must occur between all of the processes requesting use of the BBB. This arbitration may be implemented in the hardware itself or within the OS. Although processes may be denied use of the BBB for a short time, the BBB may be acquired by the next waiting process as soon as the BBB resets or the HDC saturates.

2.5 Enhancements to the base hardware

Several enhancements might be made to the base hardware. The first enhancement would be to reduce the size of the BBB by considering only conditional and indirect branches. The execution and taken profile weights of blocks with unconditional branches or direct calls can be determined by inference via Kirchhoff's First Law and need not be profiled. However, this approach requires that the runtime optimizer spend more time analyzing and constructing the control flow graph.

A second enhancement would be to index into the BBB with a combination of the branch's address and its direction. Thus, the taken and not taken paths of a particular branch would be recorded in separated entries allowing for the elimination of the taken counter in each BBB entry. While this approach would result in either an increase in the size of the BBB or a reduction in the number of distinct branch addresses in the BBB, it would allow the system to detect finer changes in program, and hot spot, behavior. For example, control flow changes within a particular set of blocks would now be detected and possibly reoptimized.

A third enhancement would be to include support for profiling the arc weights of indirect branches. Currently, profiling determines only the branch weights of these branches. An additional table indexed on a combination of the branch and target addresses could be used to store the actual profile. The BBB entry for that branch would still gather the branch execution count and determine branch candidacy.

A fourth enhancement would be to add a secondary, coarse-grained component to the Monitor Table. This secondary table would track ranges of addresses rather than single branches. Each range table entry would consist of a base address and size, and any address falling inside the range would be considered part of a hot spot. Since a hot spot would be returned from the optimizer as a single, contiguous region with a larger code size due to aggressive optimizations, the region could be added to the range table and thus help avoid conflicts among entries in the fine-grained table. Clearly, this enhancement would require a higher degree of operating system support to manage.

Benchmark	Num. Insts.	Actions Traced
099.go	89.5M	2stone9.in training input
124.m88ksim	120M	clt.in training input
126.gcc	1.18B	amptjp.i training input
129.compress	2.88B	test.in training input *count* enlarged to 800k
130.li	151M	train.lsp training input (6 queens)
132.ijpeg	1.56B	vigo.ppm training input
134.perl	2.34B	jumble.pl training input
147.vortex	2.20B	vortex.in training input
MSWord(A)	325M	opened 16.0 MB .doc file, searched for number, then closed
MSWord(B)	912M	loaded 25 page .doc file, repaginated ran built-in word count, selected entire document, changed font to Arial, undo change midway through, closed file
MSExcel	160M	generated silicon diffusion data from VB script & graphs from data
Adobe Photo-Deluxe(A)	390M	loaded detailed tiff image, brightened, increased contrast,and saved as PhotoDeluxe image
Adobe Photo-Deluxe(B)	108M	exported detailed tiff image to encapsulated postscript
Ghostview	3.17B	loaded ghostview, loaded 9 page ps file, viewed and zoomed in on 4 pages, performed text extract

Table 1. Benchmarks used for hot spot detection experiments.

3 Experimental Evaluation

Trace-driven simulations were performed for a number of benchmarks in order to explore hot spot characteristics and to establish the effectiveness of the proposed hot spot detection scheme. Both *SPECINT95* and common *WindowsNT* applications were simulated to provide a broad spectrum of typical programs. These benchmarks are summarized in Table 1. The eight applications from the SPECINT95 benchmark suite were compiled from source code using the *Microsoft VC++ 6.0* compiler with the *optimize for speed* and *inline where suitable* settings. Several WindowsNT applications executing a variety of tasks were also simulated. These applications are the general distribution versions, and thus were compiled by their respective software vendors. In order to ensure examination of all executed user instructions, sampling was not used

Parameter	Setting
Num BBB entries	2048
BBB associativity	2-way
Exec and taken cntr size	9 bits
Candidate branch thresh	16
Refresh timer interval	4096 branches
Clear timer interval	65535 branches
Global cand cntr size	13 bits
Global cand cntr inc	2
Global cand cntr dec	1
Global mon cntr size	12 bits
Global mon cntr inc	1
Global mon cntr dec	1

Table 2. Hardware parameter settings.

during trace acquisition or simulation.

The experiments were performed using the inputs shown in Table 1. The hot spot detection hardware was then simulated on an instruction-by-instruction basis for the entire execution. In order to extract complete execution traces of these applications (all user code, including statically- and dynamically-linked libraries), we employed special hardware capable of capturing dynamic instruction traces on an AMD K6 platform. This unit, known as SpeedTracer, was donated by AMD for our research.

3.1 Hardware Parameters

Because the design space is large, experimentally evaluating the individual effect of each hardware parameter was infeasible. We, therefore, selected initial parameters that attempted to match the observed hot spot behavior and then further refined them, resulting in parameters that exhibit desirable hot spot collection behavior. These parameters were used in the experiments presented in this section and are shown in Table 2. The BBB hardware was configured to allow branches with a dynamic execution percentage of .4% (16 executions/4096 branches) or higher to become candidates (the candidate ratio). The HDC was configured to require that the candidate branches total more than 66% (2:1) of the execution to become a hot spot (the threshold execution percentage). The BBB was allowed 16 refreshes (totaling 65535 branches) to detect a hot spot before it was reset. To collect these results, we used a Monitor Table large enough to contain all hot spot branches in order to examine the hot spot behavior over the course of the whole execution.

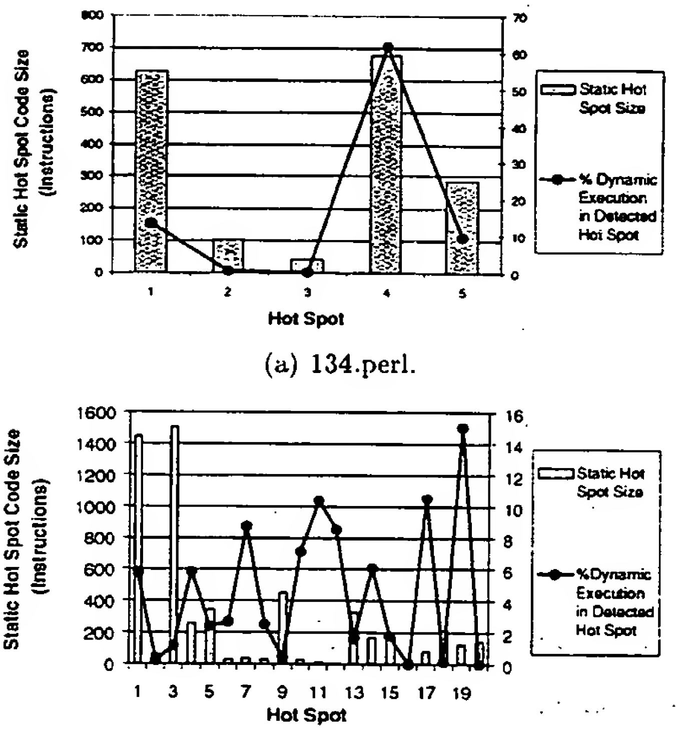
3.2 Results

Table 3 summarizes the effectiveness of our proposed hardware at detecting runtime optimization opportunities for each benchmark. The *number of hot spots* column lists the number of times that the HDC saturated at zero, indicating the detection of a new hot spot. This number also represents the number of times that the operating system was interrupted by the BBB. The *number of static instructions in hot spots* is a close estimate of the total number of instructions that will be delivered to the optimizer over the entire execution. The next column, *percent static executed instructions in hot spots*, relates the number of static instructions in hot spots to the total number of static instructions executed. In other words, out of all the static instructions executed by the microprocessor, only a small percentage lie within hot spots. The portion of total dynamic instructions represented by these hot spots is shown in the next column, *percent total execution in hot spots*. Because this hardware cannot detect hot spots instantly, some time that could be spent executing in optimized hot spots is spent during detection. The time spent in hot spots after they are detected is shown in the *percent total execution in detected hot spots*, and the time lost to detection can be found by taking the difference between this column and the previous column. Finally, the last column, *dynamic instructions in hot spots after detection*, shows the number of dynamic instructions that could benefit from runtime optimization. This number reflects any subsequent reuses of detected hot spots. For the purposes of our experimentation, when an instruction is part of several hot spots, we heuristically attribute the instruction to the most recently accessed of the relevant hot spots.

Analysis of the results shows that only a small percentage, usually less than 3%, of the static code seen by the microprocessor executes intensively enough to become hot spots. Since a large percentage of the dynamic execution is represented by a small set of instructions, often nearly 90% of the program's execution, a runtime optimizer could easily focus on this small set with the potential for significant performance increase. In addition, only about 1% of the possible time spent in optimized hot spots is lost due to detection. For example, in 130.li, the number of hot spot static instructions comprise only 3.00% of the total static instructions, yielding a total hot spot code size of 1447 instructions. Furthermore, 90.88% of the entire execution is spent in detected hot spots. Our analysis shows that ideally the hot spots account for 91.28%, and, thus, only .40% is lost during the detection process. This indicates that our Hot Spot Detector makes the identification so swiftly that the execution of hot spot regions falls almost entirely within potentially runtime optimized code.

Benchmark	# hot spots	# static insts. in hot spots	% static executed insts. in hot spots	% total exec. in hot spots	% total exec. in detected hot spots	Dyn. insts. in hot spots after detection
099.go	6	2398	3.46	37.84	35.39	31.7M
124.m88ksim	4	1576	2.78	93.03	92.30	110M
126.gcc	47	17665	8.90	58.42	52.12	617M
129.compress	7	918	2.12	99.93	99.81	2.87B
130.li	8	1447	3.00	91.28	90.88	137M
132.ijpeg	8	2556	3.48	91.07	91.00	1.42B
134.perl	5	1738	2.13	88.43	85.99	2.01B
147.vortex	5	2161	1.76	72.30	71.93	1.58B
MSWord(A)	5	3151	1.17	91.36	91.08	296M
MSWord(B)	21	12541	2.40	69.13	62.04	566M
MSExcel	25	18936	2.94	60.01	54.85	88.2M
PhotoD.(A)	20	5485	1.68	94.31	90.97	354M
PhotoD.(B)	14	4192	1.78	94.24	90.81	98.5M
Ghostview	33	8938	2.82	73.39	72.55	2.30B

Table 3. Summary of the hot spots found in the benchmarks.



(a) 134.perl.

(b) PhotoDeluxe(A).

Figure 6. Detailed hot spot statistics.

Examining individual hot spots reveals interesting characteristics of program behavior. Figure 6(a) details the detected hot spots from the 134.perl benchmark. For each hot spot, the bar graph shows the static code size of the hot spot, while the line graph shows the percentage of the execution spent in that hot spot after detection. This benchmark consists of three primary hot spots: hot spots 1, 4, and 5 on the graph. These correspond to the three hot spots in Figure 1 in the introduction (note that in the histogram, 134.perl was compiled for the *IMPACT* architecture without inlining). From the histogram, code can be seen executing between the first and second primary hot spots. Hand analysis did not classify code in that region to be hot spots, but the hardware did as actual hot spots 2 and 3. Upon further hand examination of actual hot spot 3, an intensely executed region of code is found, thus verifying the hardware's determination. In hot spot 3, the *cmd_exec* function loops 117k times calling *str_free* in each iteration. The 9 blocks, totaling 43 static instructions, contribute 7.3M dynamic instructions to the program's total execution. Because of the intense nature of these blocks, they serve as good candidates for runtime optimization. Analysis of this benchmark also shows that one hot spot is much more dominant than the others in terms of dynamic execution. In this case, optimizing only hot spot 4 could benefit over 58% of the dynamic instructions executed.

Similar characteristics were observed in the other benchmarks. Figure 6(b) shows an example from one of the pre-compiled WindowsNT applications. For this benchmark, there are a few hot spots that each represent 8% or more of the total execution which together represent more than 50%. We also see quite a few hot spots with small static code sizes, indicating tight, intensely executed code. In fact, for this benchmark, the smaller-sized hot spots are also those with high total execution percentages, indicating excellent opportunities for runtime optimization.

The benchmark 099.go is a notable example of a benchmark without obvious hot spots. While this game simulation repetitively executes players' moves, each move touches a large amount of static code with little temporal repetition. The hardware was still able to detect six hot spots representing 35% of the execution. There is one primary hot spot that represents 28% of the execution with a static code size of 1170 instructions.

Our data has shown that the static sizes of the detected hot spots vary significantly, from tens of instructions to the low thousands. At this point in the development of runtime optimization technology, the maximum static code size that a runtime optimizer could handle is an open question. The goal of our work is to quickly and accurately identify the hot spots, without placing too many artificial restrictions on the characteristics of the hot spots detected. We believe that it is better to submit a larger region of code to the runtime optimizer and rely on the optimizer to pare down the code size using all available information that has been gathered about the hot spot. However, by adjusting the existing hardware parameters it is easy to limit the size of the hot spots detected.

The dynamic execution lengths of the individual hot spots also vary significantly. In some cases, the execution lengths of the hot spots are fairly small but still account for hundreds of thousands of instructions. These lengths may be too short to benefit from runtime optimization, especially when the performance gains earned from the optimized code must offset the cost of detection and reoptimization. Unfortunately, the total execution time is not known at runtime. The operating system will have to make further decisions about which hot spots to actually optimize.

3.3 Implementation Cost

For our experiments we used a BBB large enough to guarantee accurate results for all of our benchmarks. With 2048 entries available, very few cache conflicts occurred, and 9-bit execution and taken counters were sufficient to capture profile data for most hot spots. For an equivalent hardware implementation, each BBB entry would contain a 22-bit tag field, two 9-bit counters, and a one-bit candidate flag. Thus, the total BBB hardware cost is slightly over 10 kilobytes. Compared to AMD's K7, which is reported to have a 128KB on-chip L1 cache, this size seems justifiable. Although a variety of hardware schemes may be used to implement the Monitor Table, it is likely to require less hardware than the BBB. A Monitor Table (without a coarse-grained component) that accommodates 3000 branch entries would achieve ideal results for our benchmarks. The Monitor Table, however, does not require profile counters, and, furthermore, a clever implementation may use a single entry to represent several nearby branches.

Next, we considered the cycles lost to reporting the hot spots to the OS. These cycles offset some of the gains made by executing in optimized code. The operating system is only interrupted when a hot spot is reported by the BBB. A rough estimate can be made about the time required to transfer the contents of the BBB to operating system memory. If we conservatively suppose that each of the BBB's 2048 41-bit entries requires three cycles to transfer, then the net cost per hot spot is 6144 cycles. Even for 126.gcc with 47 hot spots, the net cost is 288K cycles, which is negligible when compared to the number of instructions (617M) spent executing within the hot spots.

4 Related work

Several strategies have been proposed to collect profile data, and ideas from those implementations have motivated our efforts to create the Branch Behavior Buffer. Other recent efforts, however, have focused on estimating average program behavior through statistical sampling. Consequently these techniques rely on heuristics used during static analysis to discover important regions of code. To our knowledge only our method addresses the unique requirements of runtime optimization.

Conte, Menezes, and Hirsch [13] also propose the use of dedicated hardware for profiling. Their profile buffer is in the retirement stage and interacts with the reorder buffer to determine whether a branch has been taken. The operating system periodically samples the profile buffer, gradually constructing an arc-based profile for use in static compiler optimizations. Various techniques are explored to reduce contention among branches within the profile buffer, and similar schemes could be used to lower the hardware cost of our Branch Behavior Buffer.

The *ProfileMe* approach taken by Dean et al [14] collects detailed information about a single sampled instruction and the interaction between a pair of sampled concurrent instructions. They capture interesting events such as cache misses and branch mispredictions so that an in-depth performance analysis of the microarchitecture can be performed offline.

The Morph system, developed by Zhang et al [3] uses operating system and compiler technology to provide a framework for continuous profile-driven optimization. Although our proposed hardware greatly reduces the demand for software support, a low-overhead OS layer is still necessary. Because Morph demonstrates the feasibility of such an OS infrastructure, it is orthogonal to our hot spot detection mechanism.

Although not the focus of this paper, we envision utilizing our binary reoptimization technology and implementing region-based runtime optimizations [15] using our hot spot profile data. Our isolation of program hot spots also lends itself to the hot-cold optimization strategy proposed by Cohn and Lowney [16]. By excluding the infrequently used, or cold, instructions and concentrating on the intensely used, or hot, instructions in a hot spot, we can potentially derive significant performance benefit.

5 Conclusion

Traditional profiling focuses on collecting data to be used for compiler optimizations in a compile-run-recompile methodology. This approach restricts aggressive optimizations to a specific workload that is assumed to be representative. We have presented a method that dynamically identifies hot spots within a program and collects sufficient data to provide a framework for timely runtime optimization during the same execution. Our scheme requires minimal operating system support, incurs negligible runtime overhead, and requires a few hardware tables and counters located off of the critical execution path to identify and monitor the hot spots.

Detailed trace-driven simulations were performed for a number of integer programs, including benchmarks from SPEC95 and several WindowsNT applications. The resulting data presented in this paper showed that the hot spot detection algorithm, when applied to many common benchmarks and applications, achieves a high rate of success and provides a promising framework for building a runtime optimizer for general purpose programs. Subsequent efforts will be directed toward implementing several region-based runtime optimizations using the hot spot data and further exploring methods of delineating hot spots in the code.

6 Acknowledgments

The authors would like to thank all the members of the IMPACT team for their valuable comments, along with Microsoft for the donation of software tools. This research has been supported by Advanced Micro Devices under the direction of Dr. Dave Christie.

References

[1] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?," in *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pp. 1–14, October 1997.

[2] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming*, vol. 24, pp. 187–206, April 1996.

[3] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," in *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pp. 15–26, October 1997.

[4] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitve profiling," *ACM SIGPLAN Notices*, vol. 32, pp. 85–96, June 1997.

[5] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 259–269, December 1997.

[6] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, "Fast, effective dynamic compilation," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, vol. 31, pp. 149–159, June 1996.

[7] T. Kistler, "Dynamic runtime optimization," in *Proceedings of the Joint Modular Languages Conference*, pp. 53–66, 1997.

[8] B. L. Deitrich, *Static Program Analysis to Enhance Profile Independence in Instruction-Level Parallelism Compilation.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[9] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[10] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Transactions on Computers*, vol. 42, pp. 1045–1057, September 1993.

[11] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad, "Execution characteristics of desktop applications on windows nt," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 27–38, June 1998.

[12] D. Wall, "Predicting program behavior using real or estimated profiles," Tech. Rep. TN-18, Digital Equipment Corporation WRL Technical Note, Palo Alto, CA, Decemeber 1990.

[13] T. M. Conte, K. N. Menezes, and M. A. Hirsch, "Accurate and practical profile-driven compilation using the profile buffer," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 36–45, December 1996.

[14] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "Profileme: Hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 292–302, December 1997.

[15] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 158–168, December 1995.

[16] R. Cohn and P. G. Lowney, "Hot cold optimization of large windows/nt applications," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 80–89, December 1996.

G06F9/38E2S

Using Branch Handling Hardware to Support Profile-Driven Optimization

Thomas M. Conte* Burzin A. Patel* J. Stan Cox†

*Department of Electrical and Computer Engineering
University of South Carolina
Columbia, South Carolina 29205

†Database and Compiler Technology
AT&T Global Information Solutions
Columbia, South Carolina 29170

PD 30-11-94
P 12-21
10

Abstract

Profile-based optimizations can be used for instruction scheduling, loop scheduling, data preloading, function in-lining, and instruction cache performance enhancement. However, these techniques have not been embraced by software vendors because programs instrumented for profiling run 2–30 times slower, an awkward *compile-run-recompile* sequence is required, and a test input suite must be collected and validated for each program. This paper proposes using existing branch handling hardware to generate profile information in real time. Techniques are presented for both one-level and two-level branch hardware organizations. The approach produces high accuracy with small slowdown in execution (0.4%–4.6%). This allows a program to be profiled while it is used, eliminating the need for a test input suite. This practically removes the inconvenience of profiling. With contemporary processors driven increasingly by compiler support, hardware-based profiling is important for high-performance systems.

1 Introduction

Advanced compilers perform optimizations across block boundaries to increase instruction-level parallelism, enhance resource usage and improve cache performance. Many of these methods, such as trace scheduling [1], and superblock scheduling [2], either rely on or can benefit from information about dynamic program behavior. For example, traditional optimizations enhance performance by an additional 15% when combined with profile-driven superblock formation [2]. Other examples include data preloading [3], improved function in-lining [4], and improved instruction cache performance [5].

There are several drawbacks to profile-driven optimizations. Many of the techniques can result in code size explosion if they are performed too aggressively. Dynamic basic block execution frequencies can be used to reduce this phenomenon. More problematic is the task of profiling itself. Obtaining profile data through software methods can be complex and time consuming, requiring additional steps in the compilation process. The usual method employed is a *compile-run-recompile* sequence. First, the program is compiled with profiling probes placed within each basic block[1]. The program is then run using several different test inputs. The resulting profile data is used to drive a profile-based compilation of the original program.

Execution of the profiled version of the program is slow. With some methods, the profiled version runs 30 times slower than the optimized program. At best, a profiling program can be expected to run two times slower. In addition, test inputs need to be carefully chosen [6],[7].

Static estimation solves some of the problems related to gathering profile data [8]. However, these techniques are not as accurate as profiling [6],[7]. When used for superblock scheduling, static estimates achieve approximately 50% of the speedup that profiling can achieve [9].

Many commercial microprocessors, such as the Pentium series [10] and the PowerPC 604 [11], incorporate some form of branch handling hardware. This paper proposes using existing branch handling hardware, along with OS support, to obtain profile information. Using this, the slowdown for profiling is

[1] The profiling probes are extra instructions which log the execution of a basic block at run time.

MICRO 27- 11/94 San Jose CA USA
© 1994 ACM 0-89791-707-3/94/0011..$3.50

imperceptible (e.g., 0.4%–4.6% increase). This allows an application to be deployed in the field and later retrieved for profile-based recompilation. Since it captures actual usage, it solves the problem of obtaining valid test inputs for profiling. It also allows profiling of real-time applications and system software. Using dynamic information improves the accuracy of static techniques. In general, the techniques presented in this paper solve many of the problems with profiling and expand the usefulness of profile-driven optimization.

The following section reviews several hardware branch prediction mechanisms, along with published mechanisms that out-perform those currently implemented. Methods for deriving profile information from hardware are discussed in the third section. Although these methods are less accurate than full-fledged profiling, they are significantly more accurate than static estimates. Metrics to measure this error are discussed in Section 3.4. The fourth section presents experimental results and discusses the trade-offs between the various schemes. The paper closes with recommendations for hardware-supported profiling, many of which can be implemented today in existing systems.

2 Branch Prediction and Profiling

There are several contemporary dynamic branch prediction mechanisms that have been implemented in commercial processors. This section briefly reviews these schemes. A graph representation for profile information is also presented, along with two methods for grouping basic blocks into larger structures.

2.1 Contemporary branch handling mechanisms

There are two classes of branch prediction methods: *one-level* and *two-level* schemes. One-level schemes use the address of the branch instruction to index into a *branch target buffer* (BTB), which contains a small state machine for predicting the outcome of a branch. When the branch completes execution the actual outcome is used to update the state machine. Figure 1 depicts this process. The most common state machine for one-level schemes is the two-bit counter predictor, described in [12]. This predictor is implemented in several contemporary processors. The nominal size for the one-level branch prediction buffer is between 512 and 1024 entries. Our experiments show that the two-bit counter, when used with a 1024-entry BTB, achieves a branch prediction accuracy of 90% on-average across the SPEC92 benchmarks.

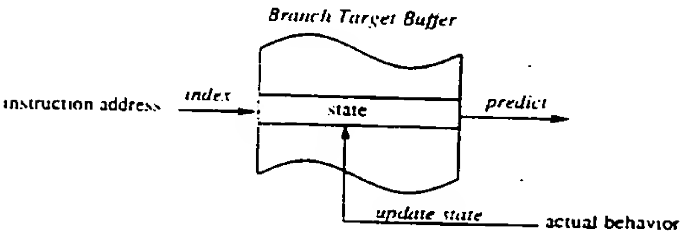


Figure 1: One-level branch prediction.

Two-level schemes use two separate buffers. The first buffer is indexed similar to the BTB and stores the branch history as a binary string. The second is indexed using this branch history and stores the state of a predictor. This is depicted in Figure 2. These schemes have been studied extensively by Yeh and Patt [13],[14] and we will use their nomenclature here. The first level buffer is termed the *history register table* (HRT). The HRT is b bits wide and stores a sequential, binary string of the branch's history, using 0 for not-taken and 1 for taken branches[2]. A prediction is made by indexing into the HRT, then using the history string to index into a second table, the *pattern table* (PT). The PT stores the state of a small state machine used to predict the branch. This decouples the branch prediction from the address of the branch instruction. The effect of this decoupling is dramatic. Yeh's algorithm can achieve 96% branch prediction accuracy for SPEC92 benchmarks [13],[14]. As of today, Yeh's algorithm has not been implemented in any commercially available microprocessor. However, the needs of wide-issue superscalars will likely drive future implementations of this branch predictor.

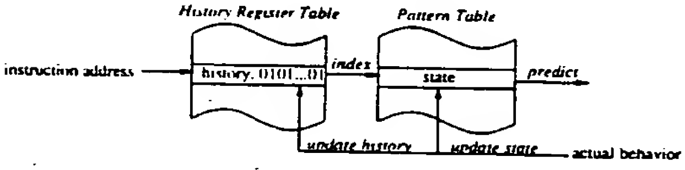


Figure 2: Two-level branch prediction.

2.2 Weighted control flow graphs

Profile-driven optimizations use a structure known as a *weighted control flow graph* (WCFG), which is a

[2] The "PAs" scheme with a 1024-entry HRT (b = 12) is used in this paper.

directed graph with basic-blocks as nodes. Arcs in a WCFG are due to one of two occurrences: either a code label or a branch instruction. An unweighted CFG for each function can be determined statically by the compiler.

A WCFG can be used to form larger groupings of blocks, which in turn can be used to enhance the scope of optimization and scheduling. Examples of these structures include Fisher's *traces* [1], and the IMPACT project *superblocks* [2]. Chang, *et al.* report a speedup of 15% when superblocks were used to extend the scope of traditional optimizations [2] Superblock formation and trace selection both use the same heuristics to form traces. Superblocks differ from traces in the method for providing fix-up code for off-trace/superblock execution and tail duplication [2],[9],[15]. Either method results in significant code size explosion. To limit this explosion, a threshold is placed on the execution frequency of a block. If a block's frequency is below this threshold, it is not considered for trace membership. (This is discussed in more detail in Section 3.4 below).

There are several methods of recording profile information. One method is to insert extra code at the beginning of each basic block that records the block id in a buffer. This buffer is then parsed into a WCFG, either periodically during execution, or after program completes execution. One example is the *Spike* profiler, which is built into the back end of GNU CC [16]. A disadvantage is its slowdown, which is approximately 30 times for *Spike*.

Another method used by AT&T Global Information Solutions in their commercial compilers is *arc-based profiling*. In this method, a transition block is added to the code to record the execution along an arc [17]. The target of the branch is changed to this new transition block, and an unconditional branch to the original destination is added to the end of the transition block. A table of all possible arcs is added to the object code by the compiler. An instruction to increment an arc's table entry is placed inside the transition block. When implemented, arc-based profiling results in a slowdown by a factor of two. Of the profiling approaches, arc-based profiling is the best suited to hardware adaptation.

2.3 The drawbacks of software profiling

Although the benefits of profile-driven optimization are large, there are many drawbacks to collecting profiles in software. The most severe is execution slowdown over unprofiled code. Slowdown is more than a minor inconvenience. Experience at AT&T Global Information Solutions has shown that slowdown is the major reason why profile-driven optimizations have not been adopted by the user community. Real-time applications such as kernels and embedded systems are excluded from the benefits of profile-driven optimizations. Long-running applications such as database systems are often excluded from profiling as well.

Another problem of profiling is the selection of inputs for the profiling task. Programs that are highly data-dependent, such as a sort routine (simple) or a database application (complex), have branches that are sensitive to user inputs. If the inputs are not selected carefully, the profile will not reflect actual usage. Validating profiling is difficult without a large scale study of user habits. In the absence of this, profiling is typically done using a large set of inputs, further increasing the time required for accurate profiling.

A third problem with profiling is the method for its use. A program must be compiled with profiling enabled, run using the test inputs, and then recompiled. For small programs, this is not difficult. For large systems, such as OS kernels or commercial database applications, this requires significant alteration of Makefile scripts [18]. A large amount of man-hours is invested in these scripts. For this reason, software vendors are hesitant to adopt any profile driven optimizations.

Ball and Larus have developed a set of heuristics to determine which arcs are more likely to be traversed in a CFG [8]. An extension to these that estimates node weights is presented in [19]. Such static heuristics can be used to solve many of the problems of profiling, but with less accurate results. For example, when static estimates are used to predict branch directions, the inaccuracies of the predictions are approximately twice that of profiled information [19].

3 Using Branch Prediction Hardware for Profiling

The goal of this paper is to demonstrate that the contents of hardware branch buffers can be used to add weights to a statically-built CFG. Most commercial processors allow the serial scan-out of state information for testing purposes. In addition to this, several processors implement kernel-mode instructions for reading branch hardware buffers directly. Hardware implementations typically include target address information along with prediction information. The combination of the target address (the destination of the arc), the buffer tag (the source of the arc), and the prediction information (the arc's weight), fully specify an arc in the WCFG.

The specific procedure for producing a WCFG is as follows: (1) A program is compiled with a special identifier token (magic number), indicating it contains a table of CFG arcs. (2) During execution, the kernel periodically reads the buffer and uses its contents to increment the arc counters. This period may be at every context switch, or more frequently. (3) On exit, the arc table is updated on disk. When branch hardware for profiling was implemented using a Pentium-based AT&T server system, results show an imperceptible difference in execution time between programs modified in this way and unmodified, traditionally-optimized programs. This slowdown is shown in Table 1 for five of the SPECint92 benchmarks. The maximum is for *espresso*, with a slowdown of just under 5%.

Table 1: Slowdown due to hardware profiling.

Benchmark	Unprofiled time (sec)	Hardware profiled time (sec)	Slow-down
compress	95.6	98.4	0.8%
eqntott	31.7	31.9	0.6%
espresso	45.4	47.6	4.6%
gcc	110.2	114.0	3.3%
xlisp	91.4	91.8	0.4%

3.1 Code adjustments to support arc-based profiling

There are two adjustments that need to be made to convert hardware branch information into arc weights. Indirect jumps can produce blocks with more than two outgoing arcs, reducing the one-to-one mapping between a buffer entry and an arc. The two primary sources of indirect jumps in C are due to call-through-pointers and *switch* statements. Call-through-pointers are not problematic, since trace selection is traditionally performed on a per-function basis. *Switch* statements can be converted into a chain of *if* statements. The performance lost from this conversion is later regained when the cases of the *switch* are sorted according to execution weight. The side-effect of this conversion is an increased branch target address predictability.

The second adjustment concerns code labels. Basic blocks formed due to code-labels are never allocated entries in the hardware buffer. A solution to this problem is to use the structure of the static graph to propagate profile information to these blocks. This can be done when the program is recompiled, and does not need to be done at execution time.

3.2 Two-level profiling

Slight modifications are required to adapt two-level schemes for the recording of arc weights. Since these buffers store a history of branch behavior, counting the number of 1's in a history register and dividing by the register width can be used to estimate the weight of an arc (we will refer to this as *dumping* the history register). After a history register is dumped, it must be updated in some fashion so that its contents are not over-counted at the next dumping point. Since dumping the buffer may occur at context switch points, there is no point in preserving the contents of the history registers. For these reasons, the history registers are initialized to 0 after being dumped[3].

Zeroing the history registers does not solve the problem of over-counting an arc's weight. Entries of '0' in the registers signify not-taken (fall-through) branches. A mechanism is needed that determines which '0's are due to actual execution and which are left over from the last buffer dump. Several techniques were experimented with, including marking each history with a *dirty bit*. In the dirty bit scheme, extra bits are added, one per history entry. These bits are cleared at each dump point. When a branch indexes into a history register, the bit is set. Since the HRT is a cache-like structure, it will contain a tag store. The "dirty" state can be marked by storing an invalid, impossible tag value for the history register entry. Thus, a *dirty bit* can be implemented with little hardware modification.

For frequently-encountered branches, the *dirty bit* scheme will produce accurate results. However, if a branch is encountered infrequently, the '0' entries from the buffer dump may still remain in a register, even though the *dirty bit* is set. This can increase the error for moderate-to-lightly visited basic blocks. The results in the following section support this claim.

Another scheme is to use a *marker bit* to record the boundary between the valid and invalid branch histories, as illustrated in Figure 3. After each dump, the registers are zeroed and the LSB of each register is set to '1' (i.e., '$00\cdots01$'). As the branch updates its history, this bit shifts to the left. An extra *full bit* is maintained in the MSB of the register. When *full bit* = 1, the entire contents of the register are treated as valid at a dump point. Otherwise, only the positions to the right of the leading '1' in the register are valid (i.e., if '$00\cdots01xxxx$', then only

[3] Our experiments show that this causes negligible change in the prediction accuracy. This is because context switches will normally result in a partial or near-total flush of the buffer.

the *xxxx* bits are valid). To complete this scheme, the *full bit* is zeroed at each dump point. Note that the *marker bit* is not wasted space. The entire history register holds a valid history once the *full bit* is set. Therefore, only one additional bit per history entry is required, regardless of the length of the HRT entries.

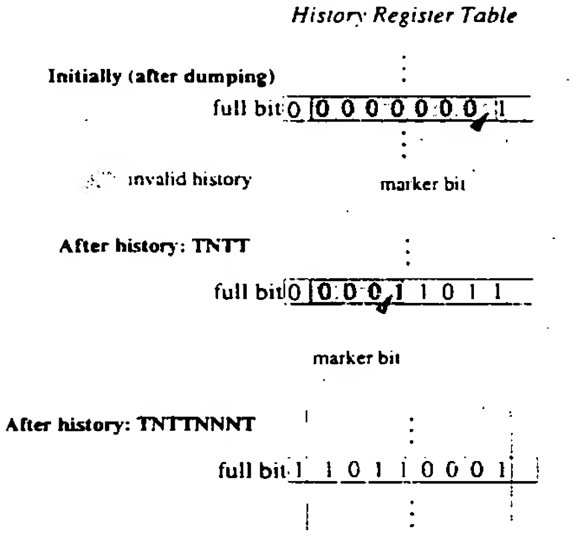

History Register Table
Initially (after dumping)
full bit 0 0 0 0 0 0 0 0 1
invalid history
marker bit
After history: TNTT
full bit 0 0 0 0 1 1 0 1 1
marker bit
After history: TNTTNNNT
full bit 1 1 0 1 1 0 0 0 1

Figure 3: The marker bit modification to Yeh's algorithm for arc weight calculation.

3.3 One-level profiling

Modification of one-level schemes is less complicated, but also less accurate. Some indicator of the validity of a branch history is still required. For one-level schemes, this can be implemented using a *dirty bit*. As with the two-level schemes, this can be implemented without any modification if the BTB maintains a tag store.

The inaccuracy for one-level schemes is a result of using a two-bit counter to estimate the number of times a branch is taken or not-taken. As with the two-level scheme, several approaches were tried until one was found that achieved highly accurate performance. This scheme relies on the ability to keep a count of the total number of instructions executed since the last history dump. This is relatively easy since most modern processors have on-chip performance monitoring hardware to record such information[4]. Given that N instructions were executed, the compiler (and hence the dumping routine) can approximate the average number of instructions per basic block, $\bar{B}$. Then, if d entries in the BTB are dirty, each entry is assumed to be touched $\hat{w} = N/(\bar{B} \times d)$ times. This value is then used to translate the two-bit counter value into increments to the arc counters. This translation is presented in Table 2. The reason for the success of these approximations is discussed in Section 4 below.

Table 2: Approximations used to convert BTB entries into arc weights.

Counter value	Value interpretation	Arc to increment	Increment value
00	strongly not-taken	fall-through	$2\hat{w}$
01	weakly not-taken	fall-through	$\hat{w}$
10	weakly taken	target	$\hat{w}$
11	strongly taken	target	$2\hat{w}$

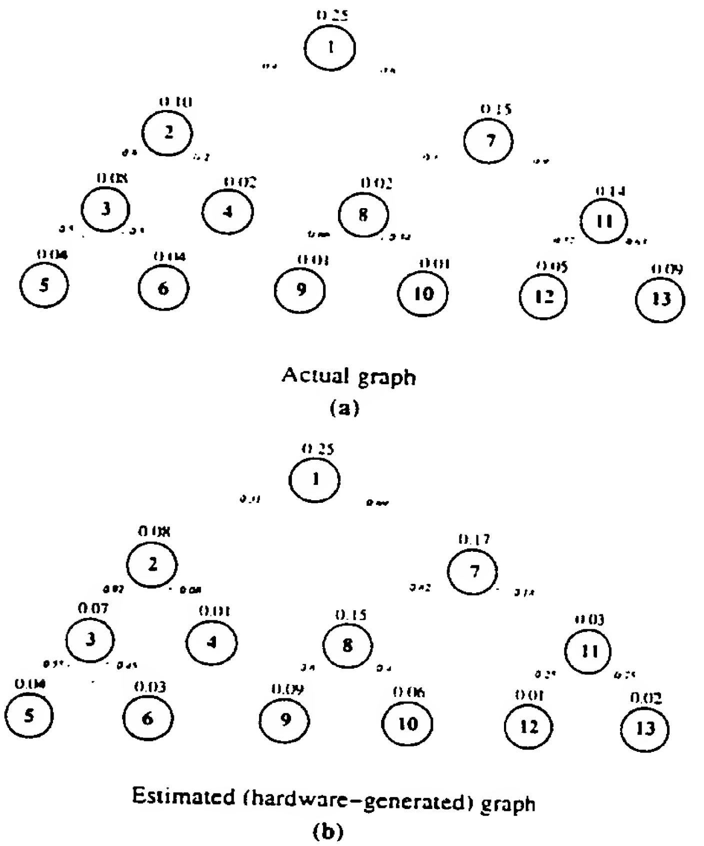
(where $\hat{w} = N/(\bar{B} \times d)$)

3.4 Comparing profiles

Validation of hardware profiling is done by comparing traditionally-generated profiles (*actual profiles*) to hardware-generated profiles (*estimated profiles*). One method for this is to perform *trace selection* on both the actual and the estimated profiles and compare the results. An example of trace selection is illustrated in Figure 4. Graph (a) is annotated with the actual profile information, whereas graph (b) is the hardware-generated profile. Traces are formed using an arc trace selection threshold of 60% to group blocks [15]. Code explosion is avoided by not extending traces to blocks with low weights. This is implemented as a threshold, T. Values of T = 0.1%, 1%, 3% and 5% are considered below.

The metric for trace selection error is introduced using the example of Figure 4. In the actual graph (graph (a)), basic blocks **1**, **7**, **11** and **13** are grouped together to form a trace. Due to errors in the weights of outgoing arcs for block **7**, the blocks **1**, **7**, **8** and **9** are grouped to form a trace in the estimated graph. The error for block **7** is due to the difference in arc weights between the two graphs. The transition from block **7** to **8** will occur 0.15 × 0.1 = 1.5% of the total execution time. Similarly, the transition from **7** to **11** will occur 0.15 × 0.9 = 13.5% of the time. (Since the actual graph contains the real execution frequencies of the program, these frequencies are used.) Hence, the transition from **7** to **11** occupies a higher percentage of the total execution. The trace in graph (b) incorrectly assumes the transition of **7** to **8** is more likely. This assumption is wrong

[4] In the absence of monitoring hardware, an approximation can be obtained using the buffer sampling rate and the processor's CPI rating.



Actual graph
(a)

Estimated (hardware-generated) graph
(b)

Figure 4: Trace selection example.

for 13.5% − 1.5% = 12% of the execution. The figure of 12% is therefore the percentage of execution time that the incorrect trace membership will be exercised.

In general, the trace selection error is the total percentage of execution time that incorrect trace membership is exercised due to errors in the estimated profile. The trace selection error (*TSE* is formally specified by taking W_i to be the normalized weight (or, execution frequency) of block i. Let w_{ij} be the normalized weight of the arc from block i to block j. The trace selection error is calculated using the following procedure:

```
1. $TSE = 0$
2. forall blocks $i \in$ WCFG(actual) do:
   (a) $j \leftarrow trace\_successor(\text{WCFG(actual)}, i)$
   (b) $k \leftarrow trace\_successor(\text{WCFG(estimate)}, i)$
   (c) if $j \neq k$ then $TSE \leftarrow TSE + W_i \times |w_{ij} - w_{ik}|$
3. enddo
```

Another method for comparison is the distribution of arc weight error versus block weights. This metric is useful since it shows where the trace selection error is occurring. The distribution is calculated by computing the maximum differences between the actual and the estimated arc weights for each category of block frequencies[5]. Let $\hat{w}_{ij}$ be the weight from i to j in the estimated (hardware-generated) profile, and w_{ij} be the weight for the actual profile. Define the maximum difference to be,

$$\Delta w_i = \max_{j \in succ(i)} |w_{ij} - \hat{w}_{ij}| . \qquad (1)$$

Then the (unnormalized) distribution function is,

$$f_{arcs}(W) = \sum_{i \text{ s.t. } W_i = W} \Delta w_i , \qquad (2)$$

or the sum of the maximum arc differences for each block with weight W. The distribution of arc weight error provides good insight into the performance of the techniques, as is shown in the next section.

4 Experimental results

The three schemes for hardware-based profiling were tested using benchmarks from the original SPEC92 benchmark suite as test workloads. Results are presented here for all the integer and an equal number of floating-point benchmarks (see Table 3). The benchmarks are compiled using the GNU C compiler with all optimizations enabled. The FORTRAN floating-point benchmarks are first converted from FORTRAN to C.

Table 3: SPEC92 benchmarks used for evaluation.

Class	Benchmark	Input
Integer	espresso	cps.in
	xlisp	li-input.lsp
	eqntott	int_pri_3.eqn
	compress	in
	sc	loada1
	gcc	tree.i
Floating-point	doduc	doducin
	nasa7	—
	mdljdp2	mdlj2.dat
	wave5	—
	tomcatv	—
	ora	params

Several approximations are made in the previous section to extract arc weights from hardware. One

[5] The maximum difference is used in order to avoid over-counting a single error. For example, there is a 4% difference for two arcs with weights 40%/60% (actual) vs. 44%/56% (estimate), not an 8% difference.

Table 4: Dynamic basic block distribution.

Bench- mark	Basic Blocks					
	E-25	E-50	E-90	E-99	E-100	Total static
espresso	15	49	225	842	2838	7582
xlisp	10	34	119	264	1058	3138
eqntott	1	2	6	34	502	1323
compress	2	5	17	21	135	432
sc	2	7	52	135	1529	4634
gcc	72	348	2610	6535	14382	34347
doduc	1	7	283	468	1596	3643
nasa7	2	2	2	2	210	1716
mdljdp2	2	5	15	35	821	848
wave5	2	14	72	177	1222	3896
tomcatv	3	6	12	14	372	1318
ora	3	6	13	24	396	1791

reason that these approximations are successful is the relatively high locality of branch instructions. This fact is illustrated in Table 4. These figures represent the distribution of unique basic blocks during execution. The "E-x" column presents the number of blocks that occupy x percent of the benchmark's execution. For example, of the 1323 branches in eqntott, only 502 are actually executed. Of these, only one branch accounts for 25% of the execution, and two branches for 50% of the execution. This table shows that most of the benchmarks exercise only a very small number of dynamic branches for the majority of their execution.

4.1 Performance of two-level profiling

Results for the two-level *dirty-bit* scheme are presented in Table 5. The columns labeled $T = y\%$ are for a code explosion cutoff threshold of $y\%$. The trace selection error is remarkably low, even for a $T = 0.1\%$ cutoff. Several of the floating-point benchmarks achieve zero error. These benchmarks have a very low number of long-life dynamic branches, as shown in Table 4. The sources of the error are explained by the distribution of arc weight error, shown in Figure 6[6]. Notice that the majority of the difference in arc weights occurs for blocks that comprise less than 1% of the execution. This is true for all the benchmarks. This shows that the majority of the error for hardware profiling occurs for the lightly-executed blocks.

The code explosion cutoff threshold has a large effect on the error. Lower cutoff thresholds produce higher error for all schemes. This is because a hardware buffer only captures the most-frequently executed branches. Seldom-executed nodes and arcs will be poorly represented in the buffer. When the threshold is 3% to 5%, the error is zero in almost all of the cases. Two exceptions are integer benchmark *compress* and the floating-point benchmark *tomcatv*. For both these benchmarks, majority of the error is due to differences in two to three arc weights between the profiles.

Table 5: Two level (dirty-bit) - trace selection error.

Benchmark	Trace selection error (percent)			
	T = 0.1%	T = 1%	T = 3%	T = 5%
espresso	16.46	2.31	0	0
xlisp	12.23	5.69	0	0
eqntott	3.95	3.64	3.64	0
compress	16.16	15.31	6.12	6.12
sc	15.37	7.36	0	0
gcc	9.09	0	0	0
doduc	2.19	0	0	0
nasa7	0.21	0	0	0
mdljdp2	3.80	2.93	0	0
wave5	5.60	4.64	0	0
tomcatv	17.87	17.87	17.87	17.87
ora	0.02	0	0	0

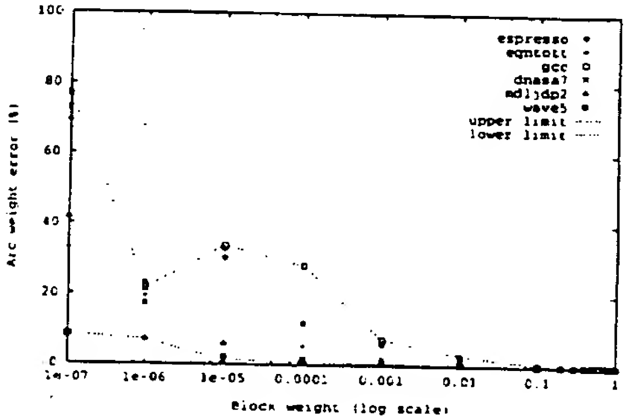


Figure 5: Two-level (*dirty-bit*): Distribution of arc weight error.

Closer examination of the error for all benchmarks suggests that the errors in trace selection have the effect of reducing the scope of the profile-based optimizations. Specifically, in *compress* a trace composed

[6] Only a subset of benchmarks are shown to simplify the graph, but their behavior is typical.

blocks 33-36-37-38 in the actual profile was split o two traces between blocks 36 and 37 in the es- ıated profile. This occurred because of an error in · estimated arc frequency between blocks 36 and . In this case the weight of this arc was less than : trace selection threshold, preventing the trace to ·w beyond beyond block 36.

[he primary reason that *tomcatv* experiences such h error is its lack of voluntary context switches ;., system calls). Because of this, the buffer is nped only when the quantum expires. In the inte- benchmarks, system calls are relatively frequent, sing a higher sampling rate. The effect of sam- ıg the buffer more frequently than once a context tch is examined below in Section 4.3. Increasing sampling rate reduces error without significant cution overhead[7].

ɔle 6: Two level (marker-bit) - trace selection error.

Benchmark	Trace selection error (percent)			
	T = 0.1%	T = 1%	T = 3%	T = 5%
espresso	12.67	2.31	0	0
xlisp	7.75	3.39	0	0
eqntott	3.95	3.64	3.64	0
compress	16.16	15.31	6.12	6.12
sc	8.46	6.40	0	0
gcc	2.39	0	0	0
doduc	1.21	0	0	0
nasa7	0.21	0	0	0
mdljdp2	3.80	2.93	0	0
wave5	2.32	2.32	0	0
tomcatv	17.87	17.87	17.87	17.87
ora	0	0	0	0

ıplementation of the *marker-bit* scheme decreases ɛ selection error over *dirty-bit* for the majority of benchmarks. For example, *espresso* drops from 6% to 12.67%. The marker improves the accu- for lightly-executed basic blocks by increasing accuracy of arc weights. This can be seen by paring the *marker-bit* arc weight error distribu- (Figure 6) to the distribution for the *dirty-bit* me (Figure 5). Observe that the error for lightly- hted blocks has been significantly reduced, es- ılly for the region between 10^{-5} and 0.001 (the ounced error crest in Figure 5).

ıe *marker-bit* scheme does not help in all ;. This is especially true when the error-causing ches executed fairly frequently. This is the case *ompress*, *espresso* and *mdljdp2*.

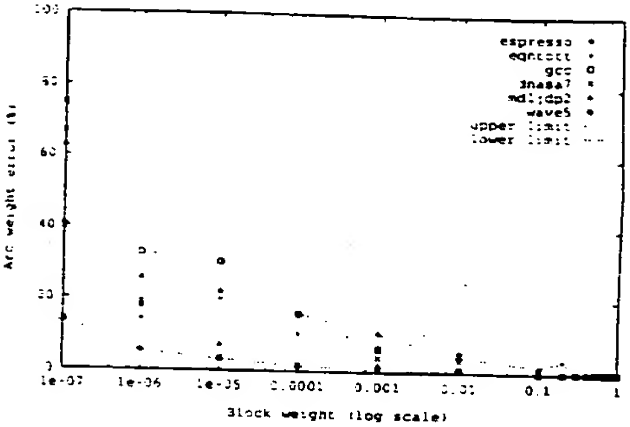


Figure 6: Two-level (*marker-bit*): Distribution of arc weight error.

4.2 Performance of one-level profiling

Profiling using one-level branch prediction hardware is often less accurate than two-level because the hardware contains a less-sophisticated measure of branch history. This results in a higher error (see Table 7). The benchmarks that perform poorly for two-level, perform poorly here as well. The arc weight error distribution (Figure 7) demonstrates why this is true. Even though the overall shape of the graph resembles the other two, there is a higher concentration of arc weight error between block weights 0.1 (10%) and 0.3 (30%). Because of this, some blocks that are relatively frequently accessed get incorrectly selected.

It is interesting to note that in some cases the one-level profiling is more accurate than the two-level schemes. This can be seen for benchmarks such as *sc*, *compress*, or *eqntott*. This is a consequence of estimating the block weight accurately. The weights are estimated using the techniques outlined in Table 2 of the previous section. This method predicts a branch's execution frequency based on how many instructions were executed since the last buffer dump. For the two-level schemes, the analog of this count is the width of the history register. This count saturates at 12, when the entire history register contains valid entries.

4.3 The effects of sampling rate on error

The results above show a relatively high error for benchmarks that sample the buffer only on quantum expiration. The worst case of this is the *tomcatv* benchmark. The effect of making the sampling rate

[7] 'he execution overhead is approximately 0.55%.

Table 7: One-level - trace selection error (percent).

Benchmark	Trace selection error			
	T = 0.1%	T = 1%	T = 3%	T = 5%
espresso	18.49	5.51	0.02	0
xlisp	8.51	2.30	0	0
eqntott	3.66	3.64	3.64	0
compress	11.92	10.89	2.65	2.65
sc	3.46	1.14	0	0
gcc	5.99	0	0	0
doduc	2.86	0.14	0	0
nasa7	0.42	0.12	0.06	0
mdljdp2	2.80	2.20	0	0
wave5	7.01	6.06	0.07	0.07
tomcatv	25.28	25.19	24.34	24.34
ora	0.02	0	0	0

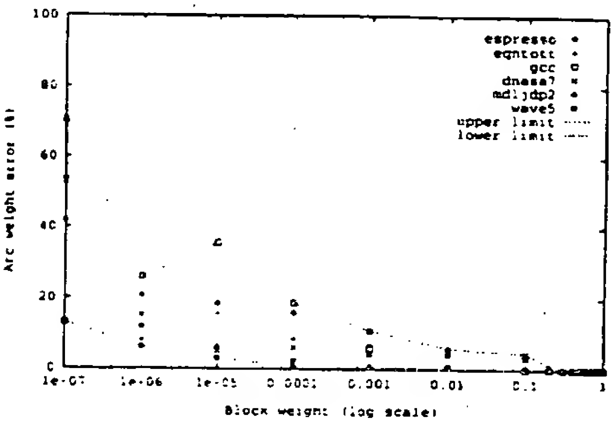


Figure 7: One-level: Distribution of arc weight error.

more frequent than the quantum are a reduction in error. This is presented in Table 8.

The error drops to zero for the two-level schemes when the sampling rate is increased to dump the buffer 16 times more frequently than the normal context switching rate. This is an important result, since it indicates a fast sampling rate can reduce even the highest error. Although the one-level approach improves with sampling rate. it does not go to zero. In general, if the scope of optimizations is limited due to trace selection error. the hardware buffer can be sampled more frequently.

Increasing the sampling rate does not appreciably affect execution time. For example. on a Pentium-based AT&T server the nominally hardware-profiled *tomcatv* takes 54.2 sec to execute. whereas interrupting *tomcatv* 16 times more frequently takes 54.5 sec to execute[8]. The reason is that a buffer dump is not a full context switch. No change of context occurs. Sampling rate can be increased without significant performance impact, provided there is kernel support. The only performance degradation comes from flushing the buffer.

Table 8: The effects of sampling rate on error (*tomcatv*).

Sampling rate	Two-level (marker-bit)			
	T = 0.1%	T = 1%	T = 3%	T = 5%
4×	17.87	17.87	17.87	17.87
8×	17.87	17.87	17.87	17.87
16×	0	0	0	0
	Two-level (dirty-bit)			
	T = 0.1%	T = 1%	T = 3%	T = 5%
4×	17.87	17.87	17.87	17.87
8×	17.87	17.87	17.87	17.87
16×	0	0	0	0
	One-level			
	T = 0.1%	T = 1%	T = 3%	T = 5%
4×	24.53	24.34	24.34	24.34
8×	19.85	19.85	17.90	17.90
16×	13.22	13.04	13.04	13.04

5 Concluding Remarks

This paper has presented a method for obtaining profile information without significant run-time slowdown (e.g., 0.4%–4.6%). This makes the ***compile-use-recompile*** approach presented here is much easier for software vendors than the traditional ***compile-run-recompile*** method of profiling. Using our techniques, software vendors can supply profiled versions of applications to alpha- and beta-testers, later collecting the profiles for final profiled optimizations. No sample suite of inputs is required. The longer the profiled version remains in the field, the higher the probability that the profiles match day-to-day use. Without the need for input sets. profiling can be used to optimize interactive, real-time, and system software packages.

All of the features required to use our techniques are already present in commercial processors. Most of these processors have branch target buffers, many that employ two-bit counter predictors. The trace selection error for these schemes is quite small. When the error does occur, it has the effect of limiting the scope of the optimization, which has few detrimental effects. It is important to note that the experimental

[8] Each of these experiments was run immediately after a reboot and the results are reproducible.

results are for a single run of each benchmark. The profiles are likely to converge after multiple runs, reducing the error still further.

Future superscalars will require branch prediction techniques more sophisticated than one-level BTB's, such as two-level approaches. Two schemes were presented to add hardware profiling to two-level mechanisms. Both schemes perform well for frequently-executed blocks. In addition, the *marker-bit* scheme performs well for moderately-executed blocks. The hardware overhead required for this scheme is minimal (specifically: one additional bit per HRT entry).

In general, the techniques presented here significantly reduce the inconvenience of profiling. With contemporary microarchitectures driven increasingly by compiler support, hardware-based profiling is important for continued improvements in processor performance.

Acknowledgements

This research has been supported by AT&T Global Information Solutions, a subsidiary of the AT&T Corporation.

References

[1] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[2] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software–Practice and Experience*, vol. 21, pp. 1301–1321, Dec. 1991.

[3] W. Y. Chen, *Data preload for superscalar and VLIW processors*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL, 1993.

[4] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling C programs," in *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, (Portland, OR), June 1989.

[5] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 242–251, May 1989.

[6] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proc. 5th Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Boston, MA), pp. 85–95, Oct. 1992.

[7] D. Wall, "Predicting program behavior using real or estimated profiles," in *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, (Toronto, Ontario, Canada), pp. 59–70, June 1991.

[8] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[9] R. E. Hank, S. A. Mahlke, J. C. Gyllenhaal, R. Bringmann, and W. W. Hwu, "Superblock formation using static program analysis," in *Proc. 26th Ann. Int'l. Symp. on Microarchitecture*, (Austin, TX), pp. 247–255, Dec. 1993.

[10] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[11] S. P. Song and M. Denman, "The PowerPC 604 RISC microprocessor," tech. rep., Somerset Design Center, Austin, TX, Apr. 1994.

[12] J. E. Smith, "A study of branch predition strategies," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, pp. 135–148, June 1981.

[13] T. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Ann. International Symposium on Microarchitecture*, (Albuquerque, NM), pp. 51–61, Nov. 1991.

[14] T. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proc. 20th Ann. International Symposium Computer Architecture*, (Ann Arbor, Michigan), pp. 257–266, May 1993.

[15] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Ann. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[16] M. L. Golden, "Issues in trace collection through program instrumentation," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1991.

[17] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," Tech. Rep. 1031, Computer Sciences Dept., University of Wisconsin-Madison, 1991.

[18] S. I. Feldman, "Make - A program for maintaining computer programs," *Software–Practice and Experience*, vol. 9, pp. 255–265, Apr. 1979.

[19] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurage static estimators for program optimization," in *Proc. 6th Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Orlando, FL), pp. 85–95, June 1994.

An Infrastructure for Profile-Driven Dynamic Recompilation*

Robert G. Burger

SAGIAN
A Division of Beckman Coulter
P. O. Box 78668
Indianapolis, IN 46278
burgerrg@sagian.com

R. Kent Dybvig

Computer Science Department
Indiana University
Lindley Hall 415
Bloomington, IN 47405
dyb@cs.indiana.edu

Abstract

Dynamic optimization of computer programs can dramatically improve their performance on a variety of applications. This paper presents an efficient infrastructure for dynamic recompilation that can support a wide range of dynamic optimizations including profile-driven optimizations. The infrastructure allows any section of code to be optimized and regenerated on-the-fly, even code for currently active procedures. The infrastructure incorporates a low-overhead edge-count profiling strategy that supports first-class continuations and reinstrumentation of active procedures. Profiling instrumentation can be added and removed dynamically, and the data can be displayed graphically in terms of the original source to provide useful feedback to the programmer.

1 Introduction

In the traditional model of program optimization and compilation, a program is optimized and compiled once, prior to execution. This allows the cost of program optimization and compilation to be amortized, possibly, over may program runs. On the other hand, it prevents the compiler from exploiting properties of the program, its input, and its execution environment that can be determined only at run time. Recent research has shown that dynamic compilation can dramatically improve the performance of a wide range of applications including network packet demultiplexing, sparse matrix computations, pattern matching, and mobile code [10, 8, 13, 16, 22, 23]. Dynamic optimization works when a program is staged in such a way that the cost of dynamic recompilation can be amortized over many runs of the optimized code [21].

This paper presents an infrastructure for dynamic recompilation of computer programs that permits optimization of code at run time. The infrastructure allows any section of code to be optimized and regenerated on-the-fly, even the code for currently active procedures. In our Scheme-based implementation of the infrastructure, the garbage collector is used to find and relocate all references to recompiled procedures, including return addresses in the active portion of the control stack.

Since many optimizations benefit from profiling information, we have included support for profiling as an integral part of the recompilation infrastructure. Instrumentation for profiling can be inserted or removed dynamically, again by regenerating code on-the-fly. A variant of Ball and Larus's low-overhead edge-count profiling strategy [2], extended to support first-class continuations and reinstrumentation of active procedures, is used to obtain accurate execution counts for all basic blocks in instrumented code. Although profiling overhead is fairly low, profiling is typically enabled only for a portion of a program run, allowing subsequent execution to benefit from optizations guided by the profiling information without the profiling overhead.

A side benefit of the profiling support is that profile data can be made available to the programmer, even as a program is executing. In our implementation, the programmer can view profile data graphically in terms of the original source (possibly split over many files) to identify the "hot spots" in a program. The source is color-coded according to execution frequency, and the programmer can "zoom in" on portions of the program or portions of the frequency range.

As a proof of concept, we have used the recompilation infrastructure and profiling information to support run-time reordering of basic blocks to reduce the number mis-predicted branches and instruction cache misses, using a variant of Pettis and Hansen's basic-block reordering algorithm [24].

The mechanisms described in this paper are directly applicable to garbage-collected languages such as Java, ML,

*This material is based on work supported in part by the National Science Foundation under grant numbers CDA-9312614 and CCR-9711269. Robert G. Burger was supported in part by a National Science Foundation Graduate Research Fellowship.

0-8186-8454-2/98 $10.00 © 1998 IEEE

Scheme, and Smalltalk in which all references to a procedure may be found and relocated at run time. The mechanisms can be adapted to languages like C and Fortran in which storage management is done manually by maintaining a level of indirection for all procedure entry points and return addresses. Although a level of indirection is common for entry points to support dynamic linking and shared libraries, the level of indirection for return addresses would be a cost incurred entirely to support run-time recompilation.

Section 2 describes the edge-count profiling algorithm incorporated into the dynamic recompilation infrastructure and briefly discusses how profile data is associated with source code and presented to the programmer. Section 3 presents the infrastructure in detail and its proof-of-concept application to basic-block reordering based on profile information. Section 4 gives some performance data for the profiler and dynamic recompiler. Section 5 describes related work. Section 6 summarizes our results and discusses future work.

2 Edge-Count Profiling

This section describes a low-overhead edge-count profiling strategy based on one described by Ball and Larus [2]. Like Ball and Larus's, it minimizes the total number of profile counter increments at run time. Unlike Ball and Larus's, it supports first-class continuations and reinstrumentation of active procedures. Additionally, our strategy employs a fast log-linear algorithm to determine optimal counter placement. The recompiler uses the profiling information to guide optimization, and it may also use the data to optimize counter placement, as described in Section 3. The programmer can view the data graphically in terms of the original source to identify the "hot spots" in a program.

Section 2.1 summarizes Ball and Larus's optimal edge-count placement algorithm. Section 2.2 presents modifications to support first-class continuations and reinstrumentation of active procedures. Section 2.3 describes our implementation. Section 2.4 explains how the profile data is correlated with the original source.

2.1 Background

Figure 1 illustrates Ball and Larus's optimal edge-count placement algorithm using a Scheme procedure that removes all occurrences of a given item from a list.

A procedure is represented by a control-flow graph composed of basic blocks and weighted edges. The assembly language instructions for the procedure are split into basic blocks, which are sequential sections of code for which control enters only at the top and exits only from the bottom. The branches at the bottoms of the basic blocks determine how the blocks are connected, so they become

```
(define remq
  (lambda (x ls)
    (cond
      [(null? ls) '()]
      [(eq? (car ls) x) (remq x (cdr ls))]
      [else (cons (car ls) (remq x (cdr ls)))])))
```

```
      bne   arg2, empty-list, L1        B1
      ld    ac, empty-list              B2
      jmp   ret
L1:   ld    ac, car-offset(arg2)        B3
      bne   ac, arg1, L2
      ld    arg2, cdr-offset(arg2)      B4
      jmp   reloc remq
L2:   st    arg2, 4(fp)                 B5
      st    ret, 0(fp)
      ld    arg2, cdr-offset(arg2)
      ld    ret, L3
      add   fp, 8, fp
      jmp   reloc remq
L3:   sub   fp, 8, fp                   B6
      ld    arg2, 4(fp)
      ld    ret, 0(fp)
      ld    arg1, ac
      ld    ac, car-offset(arg2)
      sub   ap, car-offset, xp
      add   ap, 8, ap
      st    ac, car-offset(xp)
      st    arg1, cdr-offset(xp)
      ld    ac, xp
      jmp   ret
```

```
> (remq 'b '(b a b c d))
|(remq b (b a b c d))
|(remq b (a b c d))
| |(remq b (b c d))
| |(remq b (c d))
| | |(remq b (d))
| | | |(remq b ())
| | | |()
| | |(d)
| |(c d)
|(a c d)
(a c d)
```

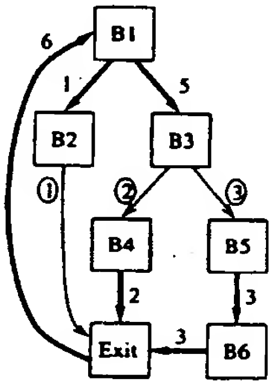


Figure 1: *remq* source, basic blocks, sample trace, and control-flow graph with thick, uninstrumented edges from one of the twelve maximal spanning trees and encircled counts for the remaining, instrumented edges.

```
(define fact
  (lambda (n done)
    (if (< n 2)
        (done 1)
        (* n (fact (− n 1) done)))))
```

```
      bge   arg1, fixnum 2, L1       B1
      ld    cp, arg2                 B2
      ld    arg1, fixnum 1
      jmp   entry-offset(cp)
L1:   st    arg1, 4(fp)              B3
      st    ret, 0(fp)
      sub   arg1, fixnum 1, arg1
      add   fp, 8, fp
      ld    ret, L2
      jmp   reloc fact
L2:   sub   fp, 8, fp                B4
      ld    arg1, 4(fp)
      ld    ret, 0(fp)
      ld    arg2, ac
      jmp   reloc •
```

Figure 2: Source and basic blocks for *fact*, a variant of the factorial function which invokes the *done* procedure to compute the value of the base case.

the edges in the graph. The weight of each edge represents the number of times the corresponding branch is taken.

The key property needed for optimal profiling is conservation of flow, i.e., the sum of the flow coming into a basic block is equal to the sum of the flow going out of it. In order for the control-flow graph to satisfy this property, it must represent all possible control paths. Consequently, a virtual "exit" block is added so that all exits are explicitly represented as edges to the exit block. Entry to the procedure is explicitly represented by an edge from the exit block to the entry block, and the weight of this edge represents the number of times the procedure is invoked.

Because the augmented control-flow graph satisfies the conservation of flow property, it is not necessary to instrument all the edges. Instead, the weights for many of the edges can be computed from the weights of the other edges using addition and subtraction, provided that the uninstrumented edges do not form a cycle. The largest cycle-free set of edges is a spanning tree. Since the sum of the weights of the uninstrumented edges represents the savings in counting, a maximal spanning tree determines the inverse of the lowest-cost set of edges to instrument.

The edge from the exit block to the entry block does not correspond to an actual instruction in the procedure, so it cannot be instrumented. Consequently, the maximal spanning tree algorithm is seeded with this edge, and the resulting spanning tree is still maximal [2].

```
> (call/cc
    (lambda (k)
      (fact 5 k)))
| (fact 5 <proc>)
| | (fact 4 <proc>)
| | | (fact 3 <proc>)
| | | | (fact 2 <proc>)
| | | | | (fact 1 <proc>)
1
```

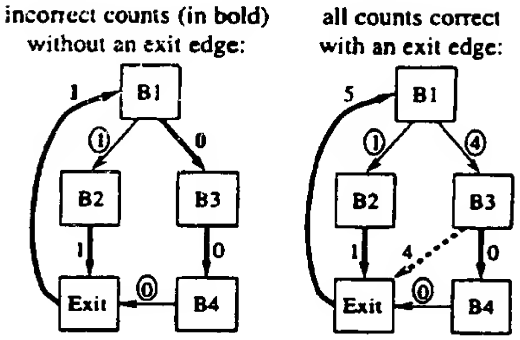


Figure 3: Trace and control-flow graphs illustrating nonlocal exit from *fact* with and without an exit edge.

2.2 Control-Flow Aberrations

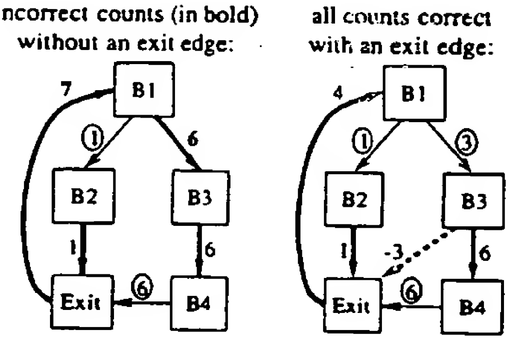
The conservation of flow property is met only under the assumption that each procedure call returns exactly once. First-class continuations, however, cause some procedure activations to exit prematurely and others to be reinstated one or more times. Even when there are no continuations, reinstrumenting a procedure while it has activations on the stack is problematic because some of its calls have not returned yet.

Figure 2 gives a variant of the factorial function that illustrates the effects of nonlocal exit and re-entry using Scheme's *call-with-current-continuation* (*call/cc*) function [11].

Ball and Larus handle the restricted class of exit-only continuations, e.g., *setjmp*/*longjmp* in C, by adding to each call block an edge pointing to the exit block. The weight of an exit edge represents the number of times its associated call exits prematurely. Figure 3 demonstrates why exit edges are needed for exit-only continuations. Ball and Larus measure the weights of exit edges directly by modifying *longjmp* to account for aborted activations. We use a similar strategy, but to avoid the linear stack walk overhead, we measure the weights indirectly by ensuring that exit edges are on the spanning tree (see Section 2.3).

With fully general continuations, the weight of an exit edge represents the net number of premature exits, and a negative weight indicates reinstatement. Figure 4 demonstrates the utility of exit edges for continuations that reinstate procedure activations.

```
> (fact 4                  | (fact 4 <proc>)
    (lambda (n)            | | (fact 3 <proc>)
      (call/cc             | | | (fact 2 <proc>)       > (redo 2)
        (lambda (k)        | | | | (fact 1 <proc>)     | | | | 2
          (set! redo k)    | | | | 1                   | | | 4
          (k n)))))        | | | 2                     | | 12
                           | | 6                       | 48
                           | 24                        48
                           24
```



incorrect counts (in bold) without an exit edge:

all counts correct with an exit edge:

Figure 4: Trace and control-flow graphs illustrating re-entry into *fact* with and without an exit edge.

2.3 Implementation

To support profiling, the compiler organizes the generated code for each procedure into basic blocks linked by edges that represent the flow of control within the procedure. It adds the virtual exit block and corresponding edges and seeds the spanning tree with the edge from the exit block to the entry block as described in Section 2.1. It then computes the maximal spanning tree and assigns counters to all edges not on the maximal spanning tree. Finally, the compiler inserts counter increments into the generated code, placing them into existing blocks whenever possible.

For efficiency, our compiler uses the priority-first search algorithm for finding a maximal spanning tree [27]. Its worst-case behavior is $O((E + B) \log B)$, where B is the number of blocks and E is the number of edges. Since each block has no more than two outgoing edges, E is $O(B)$. Consequently, the priority-first algorithm performs very well with a worst-case behavior of $O(B \log B)$.

Another benefit of this algorithm is that it adds uninstrumented edges to the tree in precisely the reverse order for which their weights need to be computed using the conservation of flow property. As a result, count propagation does not require a separate depth-first search as described in [2]. Instead, the maximal spanning tree algorithm generates the list used to propagate the counts quickly and easily. This list is especially important to the garbage collector, which must propagate the counts of recompiled procedures (see Sections 3.1 and 3.3).

If A's only outgoing edge is e, put the increment code in A.

If B's only incoming edge is e (and B is not the exit block), put the increment code in B.

Otherwise, put the increment code in a new block C that is spliced into the graph.

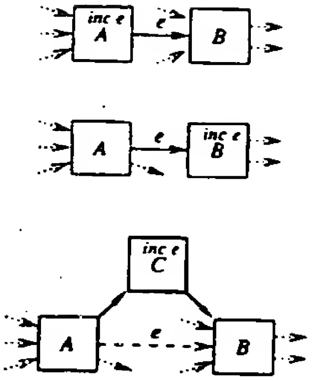


Figure 5: Efficient Instrumentation of edges.

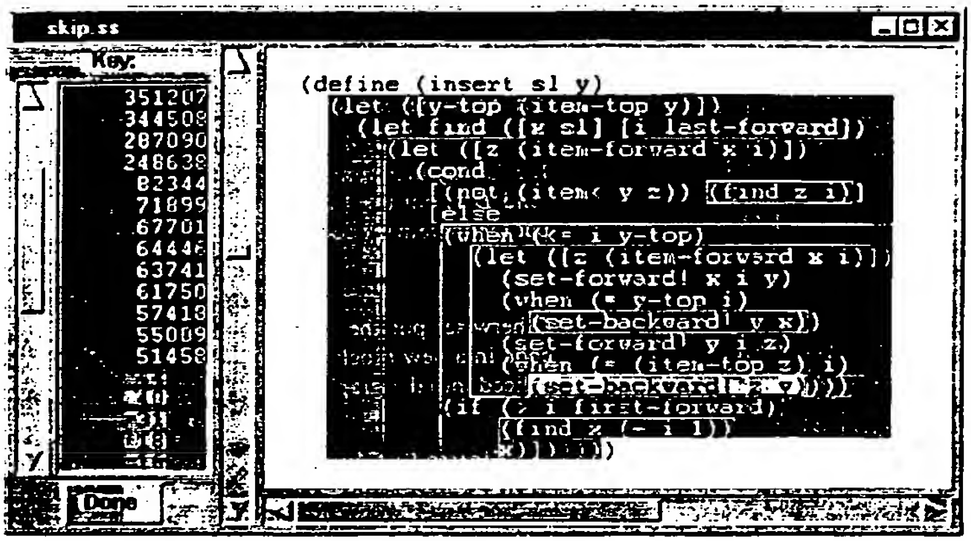
Figure 5 illustrates how our compiler minimizes the number of additional blocks needed to increment counters by placing as many increments as possible in existing blocks. The increment instructions refer to the edge data structures by actual address.

Instrumenting exit edges is more difficult because there are no branches in the procedure associated with them. We solved this problem for the edge from the exit block to the entry block by seeding the maximal spanning tree algorithm with this edge and proving that the resulting tree is still maximal. Unfortunately, exit edges rarely lie on a maximal spanning tree because their weights are usually zero. Consequently, there are two choices for measuring the weights of exit edges.

First, we could modify continuation invocation to update the weights directly. Ball and Larus use this technique for exit-only continuations by incrementing the weights of the exit edges associated with each activation that will exit prematurely [2]. This approach would support fully general continuations if it would also decrement the weights of the exit edges associated with each activation that would be reinstated. The pointer to the exit edge would have to be stored either in each activation record or in a static location associated with the return address.

Second, we could seed the maximal spanning tree algorithm with all the exit edges. The resulting spanning tree might not be maximal, but it would be maximal among spanning trees that include all the exit edges.

Our system's segmented stack implementation supports constant-time continuation invocation [17, 5]. Implementing the first approach would destroy this property. Moreover, if any procedure is profiled, the system must traverse all activations to be sure it finds all profiled ones. Although the second approach increases profiling overhead, it affects only profiled procedures, the overhead is still reasonable, and the programmer may turn off accurate continuation profiling when it is not needed. Consequently, we implemented only the second approach.



```
(define (insert sl y)
  (let ([y-top (item-top y)])
    (let find ([x sl] [i last-forward])
      (let ([z (item-forward x i)])
        (cond
          [(not (item< y z)) (find z i)]
          [else
           (when (<= i y-top)
             (let ([z (item-forward x i)])
               (set-forward! x i y)
               (when (= y-top i)
                 (set-backward! y x))
               (set-forward! y i z)
               (when (= (item-top z) i)
                 (set-backward! z y))))
           (if (> i first-forward)
               (find x (- i 1))
               x)])))))
```

Figure 6: A section of code displayed using darker shades for more frequently executed code and lighter shades for less frequently executed code.

2.4 Graphical Display of Profile Information

Although the profile data is intended primarily for use by the dynamic recompiler, it is also useful for providing feedback to the programmer. In order to provide useful feedback, some way of associating profile data with source code is needed. The compiler attaches a source record containing a source file descriptor and a byte offset from the beginning of the file to each source expression as it is read. The compiler propagates these source records through the intermediate compilation passes and associates each source record with an appropriate basic block. Within the representation of a basic block, each source record associated with the block is included within a special "source" pseudo instruction. These pseudo instructions do not result in the generation of any additional machine code.

The default location for an expression's source record is in the basic block that begins its evaluation. For procedure calls, however, the source expression corresponds to the basic block that makes the call. In most situations, the count for this block is the same as the count for the block that begins to evaluate the entire call expression. A difference arises when continuations are involved, and in this case we have found it more useful to know how many times the call is actually made.

Each constant and variable reference occurs in just one basic block, so the source record goes there. When a constant or reference occurs in nontail position, its count can usually be determined from the count of the closest enclosing expression. An exception arises when the constant or reference occurs as the "then" or "else" part of an **if** expression in nontail position. Consequently, the compiler generates source instructions for constants and references only when they occur in tail position or as the "then" or "else" part of an **if** expression. By eliminating the source instructions for the remaining cases, the compiler significantly reduces the number of source records stored in basic blocks without sacrificing useful information.

To display the block counts in terms of the original source, we follow three steps. First, we determine the count for each block by summing the weights of all its outgoing edges. Second, we build an association list of source expressions and block counts from the source records stored in each basic block. Third, we use this list to determine the source files that must be displayed by the graphical user interface and to guide a color-coding of the code according to the counts. The programmer can "zoom in" on portions of the program or portions of the frequency range and can also click on an expression to obtain its precise count. Figure 6 gives an example.

The programmer is not limited to displaying information for one procedure at a time. The data for all profiled procedures can be displayed at one time with a separate window for each source file. The data is sorted by frequency to help programmers identify hot spots. This technique proved useful in profiling the profiler itself, helping us identify inefficiencies in the maximal spanning tree and block look-up algorithms.

3 Dynamic Recompilation

This section presents the dynamic recompilation infrastructure. Section 3.1 gives an overview of the recompilation process. Section 3.2 describes how the representation of procedures is modified to accommodate dynamic recompilation. Section 3.3 describes how the garbage collector uses the modified representation to replace code objects with recompiled ones. Section 3.4 presents the recompilation process, which uses a variant of Pettis and Hansen's basic-block reordering algorithm to reduce the number of mispredicted branches and instruction cache misses [24].

3.1 Overview

Dynamic recompilation proceeds in three phases. First, the candidate procedures are identified, either by the user or by a program that selects among all the procedures in the heap. Second, these procedures are recompiled and linked to separate, new procedures. Third, the original procedures are replaced by the new ones during the next garbage collection.

Because a procedure's entry and return points may change during recompilation, the recompiler creates a translation table that associates the entry- and return-point offsets of the original and the new procedure and attaches it to the original procedure. The collector uses this table to relocate call instructions and return addresses. Because the collector translates return addresses, procedures can be recompiled while they are executing.

Before the next collection, only the original procedures are used. This invariant allows each original procedure to share its control-flow graph and associated profile counts, if profiling is enabled, with the new procedure. Because the new procedure's maximal spanning tree may be different from the original's, the new procedure may increment different counts. The collector accounts for the difference by propagating the counts of the original procedure so that the new procedure starts with a complete set of accurate counts.

3.2 Representation

Figure 7 illustrates our representation of procedures, highlighting the minor changes needed to support dynamic recompilation. A more detailed description of our object representation is given elsewhere [14].

Procedures are represented as closures and code objects. A closure is a variable-length array whose first element is the address of the procedure's anonymous entry point and whose remaining elements are the values of the procedure's free variables. The anonymous entry point is the first instruction of the procedure's machine code, which is stored in a code object. A code object has a six-word header before the machine code. The info field stores the code object's block structure and debugging information.

The relocation table is used by the garbage collector to relocate items stored in the instruction stream. Each item has an entry that specifies the item's offset within the code stream (the code offset), the offset from the item's address to the address actually stored in the code stream (the item offset), and how the item is encoded in the code stream (the type). The code pointer is used to relocate items stored as relative addresses.

Because procedure entry points may change during recompilation, relocation entries for calls use the long format so that the translated offset cannot exceed the field width. Consequently, the long format provides a convenient location for the "d" bit, which indicates whether an entry corresponds to a call instruction. Use of the long format does not significantly increase the table size because calls account for a minority of relocation entries. Because short entries cannot describe calls, they need not be checked.

Three of the code object's type bits are used to encode the status of a code object. The first bit, set by the recompiler, indicates whether the code object has been recompiled to a new one and is thus ***obsolete***. Since an obsolete code object will not be copied during collection, its relocation table is no longer needed. Therefore, its reloc field is used to point to the translation table instead. The list of original/new offset pairs is sorted by original offset to enable fast binary searching during relocation.

The second bit, denoted by "n" in the figure, indicates whether the code object is *new*. This bit, set by the recompiler and cleared by the collector, is used to prevent a new code object from being recompiled before the associated obsolete one has been eliminated. The third bit, denoted by "b" in the figure, serves a similar purpose. It indicates whether an *old* code object is *busy* in that it is either being or has been recompiled. This bit is used to prevent multiple recompilation of the same code object. The recompiler sets this bit at the beginning of recompilation. Because the recompiler may trigger a garbage collection before it creates a new code object and marks the original one obsolete, this bit must be preserved by the collector. It is possible to recompile a code object multiple times; however, each subsequent recompilation must occur after the code object has been recompiled and collected. (It would be possible to relax this restriction.)

In order to support multiple return values, first-class continuations, and garbage collection efficiently, four words of data are placed in the instruction stream immediately before the single-value return point from each non-tail call [17, 1, 5]. The live mask is a bit vector describing which frame locations contain live data. The code pointer is used to find the code object associated with a given return address. The frame size is used during garbage collection, continuation invocation, and debugging to walk down a stack one frame at a time.

3.3 Collection

Only a few modifications are necessary for the garbage collector to support dynamic recompilation. The primary

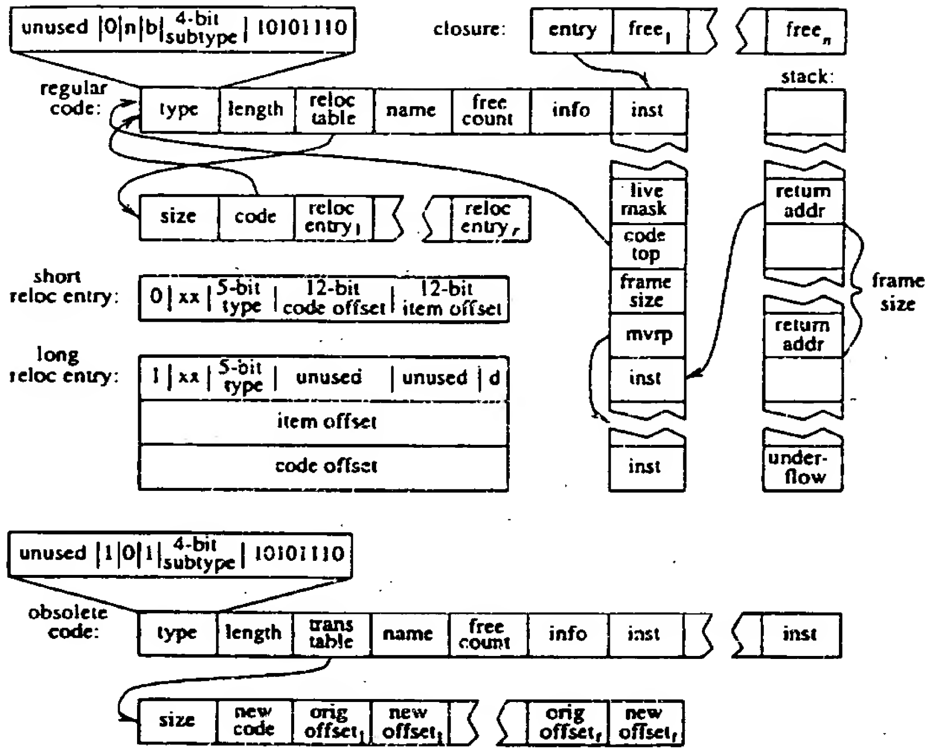


Figure 7: Representation of closures, code objects, and stacks with the infrastructure for dynamic recompilation highlighted.

change involves how a code object is copied. If the obsolete bit is clear, the code object and associated relocation table are copied as usual, but the new bit of the copied code object's type field is cleared if set.

If the obsolete bit is set, the code object is not copied at all. Instead, the pointer to the new code object is relocated and stored as the forwarding address for the obsolete code object so that all other pointers to the obsolete code object will be forwarded to the new one. Moreover, if the obsolete code object is instrumented for profiling, the collector propagates the counts so that they remain accurate when the new code object increments a possibly different subset of the counts. The list of edge/block pairs used for propagation is found in the info structure. Since the propagation occurs during collection, some of the objects in the list may be forwarded, so the collector must check for pointers to forwarded objects as it propagates the counts.

The translation table is used to relocate call instructions and return addresses. Call instructions are found only in code objects, so they are handled when code objects are swept. The "d" bit of long-format relocation entries identifies the candidates for translation. Return addresses are found only in stacks, so they are handled when continuations are swept.

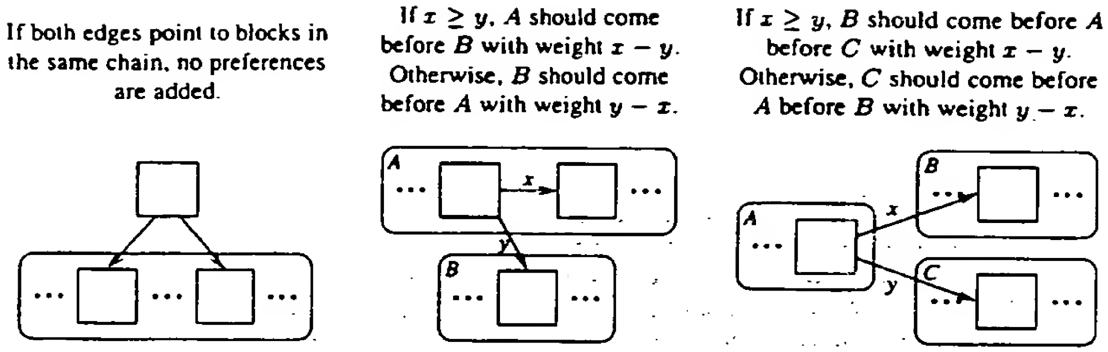
Since our collector is generational, we must address the problem of potential cross-generational pointers from obsolete to new code objects. Our segmented heap model allows us to allocate new objects in older generations when necessary [14]; thus, we always allocate new code objects in the same generation as the corresponding obsolete code objects. This allows us to keep the code space free of cross-generational pointers from older to newer objects.

3.4 Block Reordering

To demonstrate the dynamic recompilation infrastructure, we have applied it to the problem of basic-block reordering. We use a variant of Pettis and Hansen's basic-block reordering algorithm to reduce the number of mispredicted branches and instruction cache misses [24]. We also use the edge-count profile data to decrease profiling overhead by re-running the maximal spanning tree algorithm to improve counter placement.

The block reordering algorithm proceeds in two steps. First, blocks are combined into chains according to the most frequently executed edges to reduce the number of instruction cache misses. Second, the chains are ordered to reduce the number of mispredicted branches.

Initially, every block comprises a chain of one block, itself. Using a list of edges sorted by decreasing weight, dis-


If both edges point to blocks in the same chain, no preferences are added.

If $x \geq y$, A should come before B with weight $x - y$. Otherwise, B should come before A with weight $y - x$.

If $x \geq y$, B should come before A before C with weight $x - y$. Otherwise, C should come before A before B with weight $y - x$.

Figure 8: Adding branch prediction preferences for architectures that predict backward conditional branches taken and forward conditional branches not taken.

tinct chains A and B are combined when an edge's source block is the tail of A and its sink is the head of B. When all the edges have been processed, a set of chains is left.

The algorithm places ordering preferences on the chains based on the conditional branches emitted from the blocks within the chains and the target architecture's branch prediction strategy. Blocks with two outgoing edges always generate a conditional branch for one of the edges, and they generate an unconditional branch when the other edge does not point to the next block in the chain. Figure 8 illustrates how the various conditional branch possibilities generate preferences for a common prediction strategy. The preferences are implemented using a weighted directed graph with nodes representing chains and edges representing the "should come before" relation.

As each block with two outgoing edges is processed, its preferences (if any) are added to the weights of the graph. Suppose there is an edge of weight x from chain A to B and an edge of weight y from chain B to A, and $x > y$. The second edge is removed, and the first edge's weight becomes $x - y$, so that there is only one positive-weighted edge between any two nodes. A depth-first search then topologically sorts the chains, omitting edges that cause cycles. The machine code for the chains is placed in a new code object, and the old code object is marked obsolete and has its reloc field changed to point to the translation table.

4 Performance

To assess both compile-time and run-time performance of the dynamic recompiler and profiler, we used a set of benchmarks comprised of *Chez Scheme* 5.0g recompiling itself, Soft Scheme [29] checking its pattern matcher, Digital Design Derivation System 1.0 [4] deriving a Scheme CPU [6], Similix 5.0 partially evaluating itself [3], and the Gambit-C 2.3.1 benchmark suite. All measurements were taken on a DEC Alpha 3000/600 running Digital UNIX V4.0A and are reported in detail elsewhere [7].

Initial instrumentation has an average run-time overhead of 50% and an average compile-time overhead of 11%. Without block reordering, optimal count placement reduces the average run-time overhead to 37%, and the average recompile time is only 15% of the base compile time. With block reordering, the average run-time overhead drops to 27%, while the average recompile time increases very slightly to 16%.

The reduction of profiling overhead from 50% to 37% is quite respectable and shows that optimal counter placement can be an effective tool for use in long program runs intended to gather precise overall profiling information. Moreover, profile-driven run-time optimizations such as block reordering further reduce profiling overhead. Nonetheless, optimized code will run faster with profiling disabled. This suggests a strategy for dynamic optimization in which a program is dynamically optimized and profiling instrumentation removed after an initial portion of a program run during which profiling information is gathered. In addition, profiling can be enabled for particular procedures at various times during a run to provide a sort of statistical sampling, although this would likely have to be done at the programmer's direction.

To assess the effectiveness of the block-reordering algorithm, we measured the number of mispredicted conditional branches and the number of unconditional branches. The Alpha architecture encourages hardware and compiler implementors to predict backward conditional branches taken and forward conditional branches not taken [28]. Current Alpha implementations use this static model as a starting point for dynamic branch prediction. We computed the mispredicted branch percentage using the static model as a metric for determining the effectiveness of the block-reordering algorithm. Like Pettis and Hansen's algorithm, ours achieves near-optimal misprediction rates, significantly reduces the number of unconditional branches, and provides a modest 6% reduction in run time. Our algorithm is also fast, for the average recompile time is just 12% of the base compile time. Because the Alpha performs dynamic branch prediction, the 6% reduction in run time is likely due mostly to a reduction in instruction cache misses.

5 Related Work

Our edge-count profiling algorithm is based on one described by Ball and Larus [2], who applied their algorithm in a traditional static compilation environment. We have extended their algorithm to support first-class continuations and reinstrumentation of active procedures. We have also identified an algorithm for determining optimal counter placement that is faster and eliminates the need for a separate depth-first search for count propagation.

We have used Pettis and Hansen's intraprocedural block-reordering strategy with only minor modifications to apply it in the context of dynamic recompilation. Samples [26] explores a similar intraprocedural algorithm that reduces instruction cache miss rates by up to 50%. Pettis and Hansen [24] also describe an interprocedural algorithm that places procedures in memory such that those that execute close together in time will also be close together in memory. Since determining the optimal ordering is NP-complete, they use a greedy "closest is best" strategy.

Several recent research projects have focused on lightweight run-time code generation [8, 15, 22, 20, 25], with code generation costs on the order of five to 100 instructions executed for each instruction generated. Although the overhead inherent in our model is greater, the potential benefits are greater as well.

Little attention has previously been paid to profile-driven dynamic optimizations, with the notable exception of work on Self [9, 18]. This work uses a specialized form of profiling to guide generation of special-purpose code to avoid generic dispatch overhead. Our infrastructure incorporates a more general profiling strategy that is not targeted to any particular optimization technique, although we have so far applied it only to the limited problem of block reordering.

Keppel has investigated the application of run-time code generation to value-specific optimizations, in which code is special-cased to particular input values [19]. Although we have so far focused on profile-driven optimizations, we believe that our infrastructure is well suited to value-specific optimizations as well.

6 Conclusions

We have described an efficient infrastructure for dynamic recompilation that incorporates a low-overhead edge-count profiling strategy supporting first-class continuations and reinstrumentation of active procedures. We have shown how the profile data can be used to improve performance by reordering basic blocks and optimizing counter placement. In addition, we have explained how the profile data can be associated with the original source to provide graphical feedback to the programmer.

The mechanisms described in this paper have all been implemented and incorporated into *Chez Scheme*. The recompiler can profile and regenerate all code in the system, including itself, as it runs. Performance results show that the average run-time profiling overhead is initially 50% and decreases significantly with recompilation. The average compile-time profiling overhead is 11%, and the average recompile time relative to the base compile time is 12% without profiling instrumentation and 15–16% with profiling instrumentation. The block-reordering algorithm is fast and effective at reducing the number of mispredicted branches and unconditional branches.

The techniques and algorithms are directly applicable to garbage-collected languages such as Java, ML, Scheme, and Smalltalk, and can be adapted to other languages as described in Section 1.

Dynamic recompilation need not be limited to low-level optimizations such as block reordering. A promising area of future work involves associating the profile data with earlier passes of the compiler so that higher-level optimizations can take advantage of the information. For example, register allocation, flow analysis, and inlining could benefit from profile data. An unoptimized active code segment with no analogue in the optimized version of a program, such as code for an active procedure that has been inlined at its call sites, can be retained by the system until no longer needed. This will, in fact, occur naturally in garbage-collected systems.

Another area of future work involves a generalization of edge-count profiling to measure other dynamic program characteristics such as the number of procedure calls, variable references, and so forth. For example, profile data can be used to measure how many times a procedure is called versus how many times it is created, and this ratio could be used to guide lambda lifting [12]. Combined with an estimate of the cost of generating code for the procedure, this ratio could also help determine when run-time code generation [22] would be profitable. Our system can also be extended to recompile (specialize) a closure based on the run-time values of its free variables.

References

[1] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 140–149, June 1994.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[3] Anders Bondorf. *Similix Manual, System Version 5.0.* DIKU, University of Copenhagen, Denmark, 1993.

[4] Bhaskar Bose. DDD—A transformation system for Digital Design Derivation. Technical Report 331, Indiana University, Computer Science Department, May 1991.

[5] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, May 1996.

[6] Robert G. Burger. The Scheme Machine. Technical Report 413, Indiana University, Computer Science Department, August 1994.

[7] Robert G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme.* PhD thesis, Indiana University, March 1997.

[8] Chaig Chambers, Susan J. Eggers, Joel Auslander, Matthai Philipose, Markus Mock, and Przemyslaw Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *WCSSS'96 Workshop on Compiler Support for System Software*, February 1996.

[9] Craig Chambers. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, April 1992.

[10] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, July 1989.

[11] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1991.

[12] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139, 1994.

[13] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[14] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically typed languages. Technical Report 400, Indiana University, Computer Science Department, March 1994.

[15] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996.

[16] Dawson R. Engler, Deborah Wallach, and M. Frans Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science, March 1995.

[17] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[18] Urs Hölze and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–335, 1994.

[19] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[20] Mark Leone. *A Principled and Practical Approach to Run-Time Code Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. In preparation.

[21] Mark Leone and R. Kent Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Indiana University, Computer Science Department, September 1997.

[22] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[23] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Department of Computer Science, Columbia University, 1992.

[24] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[25] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A template-based compiler for 'C. In *WCSSS'96 Workshop on Compiler Support for System Software*, pages 1–7, February 1996.

[26] A. Dain Samples. Profile-driven compilation. Technical Report 627, University of California, Berkeley, 1991.

[27] Robert Sedgewick. *Algorithms*, chapter 31. Addison-Wesley Publishing Company, second edition, 1988.

[28] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[29] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, 1994.